

```

        call WriteHex
        pop     eax
    ENDM

```

mWriteSpace

宏 mWriteSpace 在控制台窗口中显示一个或多个空格，可以向该宏传递一个数字指定要显示的空格的数量（默认是 1）。例如，下面的语句将显示 5 个空格：

```
mWriteSpace 5
```

mWriteSpace 的实现：下面是 mWriteSpace 的源码。

```

;-----
mWriteSpace MACRO count:=<1>
;
; Writes one or more spaces to the console window.
; Receives: an integer specifying the number of spaces.
; Default value of count is 1.
;-----
LOCAL spaces
.data
spaces BYTE count DUP(' '),0
.code
    push    edx
    mov     edx,OFFSET spaces
    call    WriteString
    pop     edx
ENDM

```

10.3.2 节讲述了宏的参数如何使用默认的初始化值。

mWriteString

宏 mWriteString 在控制台窗口中显示一个字符串的内容。该宏简化了对 WriteString 过程的调用，允许在宏的同一行上传递字符串变量的名字。例如：

```

.data
str1 BYTE "Please enter your name: ",0
.code
mWriteString str1

```

mWriteString 的实现：下面 mWriteString 的实现在堆栈上保存 EDX 的值，然后通过 EDX 保存字符串的偏移地址，最后在过程调用后再从堆栈中弹出保存的 EDX 的值：

```

;-----
mWriteString MACRO buffer:REQ
;
; Writes a string variable to standard output.
; Receives: string variable name.
;-----
    push    edx
    mov     edx,OFFSET buffer
    call    WriteString
    pop     edx
ENDM

```

10.2.6 例子程序：封装

下面写一个小程序（Wraps.asm）来演示前面那些对过程进行了封装的宏，由于每个宏都隐藏了大量单调乏味的参数传递步骤，所以程序出奇地紧凑。到现在为止，所介绍的所有宏都在 Macros.inc 文件中定义：

```

TITLE Procedure Wrapper Macros          (Wraps.asm)
; This program demonstrates macros as wrappers
; for library procedures. Contents: mGotoxy, mWrite,
; mWriteString, mReadString, and mDumpMem.

INCLUDE Irvine32.inc
INCLUDE Macros.inc                      ; 宏定义

.data
array DWORD 1,2,3,4,5,6,7,8
firstName BYTE 31 DUP(?)
lastName  BYTE 31 DUP(?)

.code
main PROC
    mGotoxy 0,0
    mWrite <"Sample Macro Program",0dh,0ah>
; 输入用户名
    mGotoxy 0,5
    mWrite "Please enter your first name: "
    mReadString firstName
    call CrLf

    mWrite "Please enter your last name: "
    mReadString lastName
    call CrLf

; 显示用户名
    mWrite "Your name is "
    mWriteString firstName
    mWriteSpace
    mWriteString lastName
    call CrLf

; 显示数组中的整数
    mDumpMem OFFSET array, LENGTHOF array, TYPE array
    exit
main ENDP
END main

```

程序输出：下面是程序的输出示例。

```

Sample Macro Program
Please enter your first name: Joe
Please enter your last name: Smith
Your name is Joe Smith
Dump of offset 00404000
-----
00000001 00000002 00000003 00000004 00000005
00000006 00000007 00000008

```

10.2.7 本节习题

1. (真/假) 当调用宏时, CALL 和 RET 指令被自动插入到编译后的程序中。
2. (真/假) 宏的扩展是由汇编器的预处理器完成的。
3. 与使用不带参数的宏相比, 使用带参数的宏有什么优点?
4. (真/假) 只要是定义在代码段内, 宏的定义既可以在调用宏的语句之前也可以在调用宏的语句之后出现。
5. (真/假) 如果把一个长过程中的代码用宏来定义, 并且宏被调用了多次, 那么会增大编

译后程序的大小。

6. （真/假）宏不能包含数据定义。
7. LOCAL 伪指令的作用是什么？
8. 在汇编过程中哪条伪指令可以在控制台上显示信息？
9. 写一个宏 mPrintChar，在屏幕上显示一个字符。该宏应有两个参数，第一个参数指定要显示的字符，第二个参数指定字符要重复多少次。下面是调用示例：

```
mPrintChar 'X',20
```

10. 写一个宏 mGenRandom，生成一个 0 到 $n-1$ 之间的随机数， n 是唯一的参数。
11. 写一个宏 mPromptInteger，显示一个提示并读取用户输入的一个整数。该宏接收一个字符串以及一个双字变量的名字。调用示例如下：

```
.data
minVal DWORD ?
.code
mPromptInteger "Enter the minimum value", minVal
```

12. 写一个宏 mWriteAt，在控制台上定位光标并显示一个字符串。建议：调用 mGotoxy 和 mWrite 宏。
13. 列出下面调用 mWriteString 宏（10.2.5 节）的语句生成的展开代码：

```
mWriteStr namePrompt
```

14. 列出下面调用 mReadString 宏（10.2.5 节）的语句生成的展开代码：

```
mReadStr customerName
```

15. 挑战：写一个 mDumpMemx 宏，接收一个变量名作为参数。该宏必须调用 mDumpMem 宏并传递给它变量的偏移、变量的单元数目以及单元的大小。给出一个 mDumpMemx 宏的调用示例。

10.3 条件汇编伪指令

条件汇编伪指令和宏联合使用可以使得宏更加灵活，条件汇编伪指令的一般格式是：

```
IF condition
    statements
[ELSE
    statements]
ENDIF
```

本章介绍的常量条件伪指令和 6.7 节介绍的运行时伪指令（如 IF 和 ENDF）是不一样的，读者不要混淆这两者，运行时伪指令（如 IF）是基于存储在寄存器或变量中的运行时值对表达式求值的。

表 10.3 列出了一些常用的条件汇编伪指令，当说明中提到一条伪指令“允许汇编”的时候，就意味着其后直到下一个相邻的 ELSE 或 ENDF 伪指令之间的所有语句都会被编译，这里必须强调的是，表中列出的伪指令是在编译时而不是在运行时求值的。

表 10.3 条件汇编伪指令

伪 指 令	说 明
IF 表达式	如果表达式值为真（非 0）则允许汇编。可以使用的相关操作符有 LT, GT, EQ, NE, LE 和 GE

(续表)

伪 指 令	说 明
IFB <参数>	如果参数为空则允许汇编, 参数名必须用尖括号 (<>) 括起来
IFNB <参数>	如果参数不为空则允许汇编, 参数名必须用尖括号括起来
IFIDN <参数 1>,<参数 2>	如果两个参数相同则允许汇编, 参数比较是大小写敏感的
IFIDNI <参数 1>,<参数 2>	如果两个参数相同则允许汇编, 参数比较是不区分大小写的
IFDIF <参数 1>,<参数 2>	如果两个参数不同则允许汇编, 参数比较是大小写敏感的
IFDIFI <参数 1>,<参数 2>	如果两个参数不同则允许汇编, 参数比较是不区分大小写的
IFDEF 名字	如果名字已经定义则允许汇编
IFNDEF 名字	如果名字未定义则允许汇编
ENDIF	结束一个条件汇编伪指令开始的语句块
ELSE	如果前面的条件汇编伪指令的条件均为假, 则汇编该伪指令至 ENDIF 伪指令之间的语句
ELSEIF 表达式	如果前面的条件汇编伪指令的条件均为假, 则汇编该伪指令至下一个条件汇编伪指令之间的语句
EXITM	立即退出宏, 阻止其后任何语句的展开

10.3.1 检查缺少的参数

宏可以检查其任何一个参数是否为空。通常, 如果一个宏接收了空参数, 预处理器展开宏的时候就会导致生成无效指令。例如, 如果调用宏 `mWriteString` 的时候不传递参数, 在宏展开的时候, 把字符串偏移送 `EDX` 的指令就成了无效指令。下面是编译器生成的语句, 汇编器检测到了缺少参数的情况并产生一条错误信息:

```
mWriteStr
1  push edx
1  mov  edx,OFFSET
Macro2.asm(18) : error A2081: missing operand after unary operator
1  call WriteString
1  pop  edx
```

为防止缺少参数产生的错误, 可使用 `IFB` (if blank) 伪指令, 该伪指令在宏参数为空时返回真; 也可以使用 `IFNB` (if not blank) 伪指令, 它在宏参数非空时返回真。下面我们来编写一个新的 `mWriteString` 宏, 该宏在汇编过程中如果缺少参数就显示一条错误信息:

```
mWriteStr MACRO string
    IFB <string>
        ECHO -----
        ECHO *   Error: parameter missing in mWriteStr
        ECHO *   (no code generated)
        ECHO -----
    EXITM
ENDIF
push edx
mov  edx,OFFSET string
call WriteString
pop  edx
ENDM
```


(回忆一下, 10.2.2 节讲过汇编程序时 ECHO 伪指令可以在控制台上显示一条信息。) EXITM 伪指令告诉预处理器退出宏, 不要再展开宏中其后的语句。下面显示了程序在缺少宏参数的情况下汇编时产生中的屏幕输出:

```
Assembling: Macro2.asm
-----
* Error: parameter missing in mWriteString
* (no code generated)
-----
```

10.3.2 默认的参数初始化值

宏可以有默认的参数初始化值。如果调用宏时缺少参数, 则使用默认参数。格式如下所示:

```
paramname := < argument >
```

(操作符前后的空格不是必需的。)例如, mWriteLn 宏可提供一个包含一个空格的字符串作为默认参数, 如果调用时不带参数, 那么仍然会打印一个空格并在后面跟一个换行符:

```
mWriteLn MACRO text:=<" ">
    mWrite text
    call Crlf
ENDM
```

如果使用空字符串("")作为默认的宏参数, 那么编译器会产生一个错误提示, 所以至少要在两个引号之间插入一个空格。

10.3.3 布尔表达式

汇编器允许在 IF 和其他条件伪指令中包含的常量布尔表达式中使用下面的关系运算符:

LT	小于
GT	大于
EQ	等于
NE	不等于
LE	小于等于
GE	大于等于

10.3.4 IF, ELSE 和 ENDIF 伪指令

IF 伪指令后面必须跟一个常量布尔表达式。表达式可包含整数常量、符号常量或宏参数常量, 但是不能包含寄存器或变量名。其中的一种格式是只使用 IF 和 ENDIF:

```
IF expression
    statement-list
ENDIF
```

另一种格式是使用 IF, ELSE 和 ENDIF:

```
IF expression
    statement-list
ELSE
    statement-list
ENDIF
```

例子: mGotoxyConst 宏。mGotoxyConst 宏使用 LT 和 GT 操作符对传递给宏的参数进行范

围检查, 参数 X 和 Y 必须是常量。另一个符号常量 ERRS 用于统计发现的错误数, 检查 X 参数的时候, ERRS 可能被设置为 1; 检查 Y 参数的时候, 可能会再对 ERRS 加 1, 最后如果 ERRS 的值大于 0, 那么使用 EXITM 伪指令来退出宏:

```

;-----
mGotoxyConst MACRO X:REQ, Y:REQ
;
; Sets the cursor position at column X, row Y.
; Requires X and Y coordinates to be constant expressions
; in the ranges 0 <= X < 80 and 0 <= Y < 24.
;-----
    LOCAL ERRS                ;; 局部常量
    ERRS = 0
    IF (X LT 0) OR (X GT 79)
        ECHO Warning: First argument to mGotoxy (X) is out of range.
        ECHO *****
        ERRS = 1
    ENDIF
    IF (Y LT 0) OR (Y GT 24)
        ECHO Warning: Second argument to mGotoxy (Y) is out of range.
        ECHO *****
        ERRS = ERRS + 1
    ENDIF
    IF ERRS GT 0                ;; 如果发现了错误
        EXITM                  ;; 退出宏
    ENDIF
    push    edx
    mov     dh,Y
    mov     dl,X
    call    Gotoxy
    pop     edx
ENDM

```

10.3.5 IFIDN 和 IFIDNI 伪指令

IFIDNI 伪指令以大小写不敏感的方式比较两个符号 (包括宏参数) 是否相等, 如果相等的话则返回 TRUE。IFIDN 伪指令也比较两个符号是否相等, 但是要区分大小写。当要确保调用宏时传递的宏参数不和宏内使用的寄存器冲突时, IFIDNI 是很有用的, 它的使用格式如下:

```

IFIDNI <symbol>, <symbol>
    statements
ENDIF

```

IFIDN 的格式与 IFIDNI 的格式相同。

在下面的 mReadBuf 宏中, 第二个参数不能是 EDX, 因为 buffer 的偏移被送到 EDX 的时候, 参数就会被覆盖。在下面修改后的版本中, 如果不满足这个条件就显示一条错误信息:

```

;-----
mReadBuf MACRO bufferPtr, maxChars
;
; Reads from the keyboard into a buffer.
; Receives: offset of the buffer, count of the maximum
;   number of characters that can be entered. The
;   second argument cannot be edx or EDX
;-----
    IFIDNI <maxChars>, <EDX>
        ECHO Warning: Second argument to mReadBuf cannot be EDX
        ECHO *****
        EXITM
    ENDIF

```

```

ENDIF
push ecx
push edx
mov  edx,bufferPtr
mov  ecx,maxChars
call ReadString
pop  edx
pop  ecx
ENDM

```

下面的语句中由于第二个参数是 EDX，所以将导致该宏产生一条错误信息：

```
mReadBuf OFFSET buffer,edx
```

10.3.6 例子：对矩阵行求和

在 9.4.2 节已经介绍了如何对一个字节矩阵的一行求和，第 9 章的一道编程练习要求读者对这个过程进一步一般化，以处理字矩阵和双字矩阵。习题的解答可能有点冗长，下面我们看看是否可以使用宏来简化这个任务。下面是第 9 章给出的原始的 calc_row_sum 过程：

```

calc_row_sum PROC USES ebx ecx esi
;
; Calculates the sum of a row in a byte matrix.
; Receives: EBX = table offset, EAX = row index,
;           ECX = row size, in bytes.
; Returns:  EAX holds the sum.
;-----
    mul    ecx                ; row index * row size
    add    ebx,eax            ; 行的偏移地址
    mov    eax,0              ; 累加器
    mov    esi,0              ; 列的索引
L1:  movzx  edx,BYTE PTR[ebx + esi] ; 取一个字节
    add    eax,edx            ; 累加
    inc    esi                ; 行的下一个字节
    loop   L1
    ret
calc_row_sum ENDP

```

首先，把 PROC 换成 MACRO，删除 RET 指令，把 ENDP 换成 ENDM，插入等价的 PUSH 和 POP 指令替换 USES 操作符：

```

mCalc_row_sum MACRO
    push    ebx                ; 保存要修改的寄存器
    push    ecx
    push    esi
    mul    ecx                ; row index * row size
    add    ebx,eax            ; 行的偏移地址
    mov    eax,0              ; 累加器
    mov    esi,0              ; 列的索引
L1:  movzx  edx,BYTE PTR[ebx + esi] ; 取一个字节
    add    eax,edx            ; 累加
    inc    esi                ; 行的下一个字节
    loop   L1
    pop     esi                ; 恢复修改的寄存器
    pop     ecx
    pop     ebx
ENDM

```

接下来，使用宏参数替换寄存器参数并在宏内初始化寄存器参数：

```

mCalc_row_sum MACRO index, arrayOffset, rowSize
    push    ebx                ; 保存要修改的寄存器
    push    ecx

```

```

    push    esi
; 设置要求的寄存器参数
    mov     eax,index
    mov     ebx,arrayOffset
    mov     ecx,rowSize

    mul     ecx                ; row index * row size
    add     ebx,eax            ; 行的偏移地址
    mov     eax,0              ; 累加器
    mov     esi,0              ; 列的索引
L1:  movzx   edx,BYTE PTR[ebx + esi] ; 取一个字节
    add     eax,edx            ; 累加
    inc     esi                ; 行的下一个字节
    loop    L1
    pop     esi                ; 恢复修改的寄存器
    pop     ecx
    pop     ebx
ENDM

```

我们增加一个参数 `eltType`，指定数组的类型（`BYTE`、`WORD` 或 `DWORD`）：

```
mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
```

要复制到 `ECX` 中的 `rowSize` 参数现在表示的是每行的字节数，如果要以 `ECX` 作为循环计数器，它就必须包含每行的元素的数目。因此，对于 16 位的数组，把 `ECX` 除以 2；对于双字数组，把 `ECX` 除以 4。一种快速方法是以 `eltType / 2` 为计数器对 `ECX` 进行右移位操作：

```
shr ecx,(TYPE eltType / 2) ; byte=0, word=1, dword=2
```

我们在 `MOVZX` 指令的基址变址操作数中以 `TYPE eltType` 为比例因子：

```
movzx edx,eltType PTR[ebx + esi*(TYPE eltType)]
```

不过如果右边的源操作数是双字类型，`MOVZX` 无法汇编，因此必须在 `eltType` 等于 `DWORD` 的时候使用 `IFIDNI` 操作符单独创建一条 `MOV` 指令：

```
IFIDNI <eltType>,<DWORD>
    mov edx,eltType PTR[ebx + esi*(TYPE eltType)]
ELSE
    movzx edx,eltType PTR[ebx + esi*(TYPE eltType)]
ENDIF

```

最后得到了修改完成后的宏，一定要记得把标号 `L1` 修改为宏内的 `LOCAL` 标号：

```

;-----
mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
; Calculates the sum of a row in a two-dimensional array.
;
; Receives: row index, offset of the array, number of bytes
; in each table row, and the array type (BYTE, WORD, or DWORD).
; Returns: EAX = sum.
;-----
LOCAL L1
    push    ebx                ; 保存要修改的寄存器
    push    ecx
    push    esi

; 设置要求的寄存器参数
    mov     eax,index
    mov     ebx,arrayOffset
    mov     ecx,rowSize

; 计算行的偏移地址
    mul     ecx                ; row index * row size
    add     ebx,eax            ; 行的偏移地址

```

```

; 准备循环计数器
shr    ecx,(TYPE eltType / 2) ; byte=0, word=1, dword=2
; 初始化累加器和列索引
mov    eax,0                ; 累加器
mov    esi,0                ; 列的索引

L1:
    IFIDNI <eltType>, <DWORD>
        mov    edx,eltType PTR[ebx + esi*(TYPE eltType)]
    ELSE
        movzx   edx,eltType PTR[ebx + esi*(TYPE eltType)]
    ENDIF
    add    eax,edx            ; 取一个字节
    inc    esi
    loop   L1

    pop    esi                ; 恢复修改的寄存器
    pop    ecx
    pop    ebx
ENDM

```

下面是对该宏的调用示例，示例中使用了字节数组、字数组和双字数组，参见 rowsum.asm 程序：

```

.data
tableB    DWORD    10h, 20h, 30h, 40h, 50h
RowSizeB  = ($ - tableB)
          DWORD    60h, 70h, 80h, 90h, 0A0h
          DWORD    0B0h, 0C0h, 0D0h, 0E0h, 0F0h

tableW    DWORD    10h, 20h, 30h, 40h, 50h
RowSizeW  = ($ - tableW)
          DWORD    60h, 70h, 80h, 90h, 0A0h
          DWORD    0B0h, 0C0h, 0D0h, 0E0h, 0F0h

tableD    DWORD    10h, 20h, 30h, 40h, 50h
RowSizeD  = ($ - tableD)
          DWORD    60h, 70h, 80h, 90h, 0A0h
          DWORD    0B0h, 0C0h, 0D0h, 0E0h, 0F0h

index DWORD ?
.code
mCalc_row_sum index, OFFSET tableB, RowSizeB, BYTE
mCalc_row_sum index, OFFSET tableW, RowSizeW, WORD
mCalc_row_sum index, OFFSET tableD, RowSizeD, DWORD

```

10.3.7 特殊操作符

有 4 个特殊的汇编操作符可使宏的使用更加灵活：

&	替换操作符
<>	文本操作符
!	特殊字符操作符
%	展开操作符

替换操作符(&)

替换操作符(&)将宏内部对宏参数的引用替换为调用时的实际值，这在对宏参数的引用可能有歧义的情况下是特别有用的。例如 mShowRegister 宏(10.2.5 节)显示 32 位寄存器的名称及其内容(以十六进制值显示)，下面是调用示例：

```

.code
mShowRegister ECX

```

下面是调用该宏产生的输出:

```
ECX=00000101
```

在宏的内部可以定义一个包含寄存器名的字符串变量:

```
mShowRegister MACRO regName
.data
tempStr BYTE " regName=",0
```

但预处理器会认为 `regName` 是字符串的一部分,并不会用传递给宏的实际参数替换它。如果在它前面加一个 `&` 操作符,就会强制预处理器在字符串中插入实际传递的宏参数(例如 `ECX`)。下面演示了如何正确地定义 `tempStr`:

```
mShowRegister MACRO regName
.data
tempStr BYTE " &regName=",0
```

展开操作符 (%)

展开操作符 (%) 展开文本宏或把常量表达式转换成文本表示形式。这有几种情况,在与 `TEXTEQU` 联合使用时, % 操作符对常量表达式求值并把结果转换成整数。在下例中, % 操作符对表达式 `(5+count)` 求值并返回整数 15 (作为文本):

```
count = 10
sumVal TEXTEQU %(5 + count)      ; = "15"
```

如果宏要求使用整数常量参数, % 操作符允许灵活地传递整数表达式。表达式首先被计算求值得出整数结果,然后再传递给宏。例如下例中调用 `mGotoxyConst` 宏时,其参数中的两个表达式分别被求值为 50 和 7:

```
mGotoxyConst %(5 * 10), %(3 + 4)
```

预处理器将生成下面的语句:

```
1  push edx
1  mov  dh,7
1  mov  dl,50
1  call Gotoxy
1  pop  edx
```

行首的 %: 当展开操作符 (%) 作为源代码行的第一个字符的时候,就指示预处理器展开在该行发现的所有文本宏和宏函数。例如,假设在汇编过程中要在屏幕上显示数组的大小,下面的语句并不会得到期望的结果:

```
.data
array DWORD 1,2,3,4,5,6,7,8
.code
ECHO The array contains (SIZEOF array) bytes
ECHO The array contains %(SIZEOF array) bytes
```

屏幕上的输出毫无用处:

```
The array contains (SIZEOF array) bytes
The array contains % (SIZEOF array) bytes
```

如果使用 `TEXTEQU` 创建一个包含 `(SIZEOF array)` 的文本宏,就可以在下一行中使用 % 操作符来将其展开:

```
TempStr TEXTEQU %(SIZEOF array)
% ECHO The array contains TempStr bytes
```

产生的输出如下:

```
The array contains 32 bytes
```

显示行号: 下面的 Mul32 宏把前两个参数相乘并在第三个参数中返回乘积, 该宏的参数可以是寄存器、内存操作数甚至是立即操作数 (除了第三个参数之外):

```
MUL32 MACRO op1, op2, product
    IFIDNI <op2>, <EAX>
        LINENUM TEXTEQU %(@LINE)
        ECHO -----
%    ECHO *   Error on line LINENUM: EAX cannot be the second
    ECHO *   argument when invoking the MUL32 macro.
        ECHO -----
    EXITM
    ENDIF
    push eax
    mov  eax, op1
    mul  op2
    mov  product, eax
    pop  eax
ENDM
```

Mul32 宏要求第二个操作数不能是 EAX。有趣的是, 这个宏显示了调用宏的语句所在的行号, 以利于跟踪并修正问题。在这个宏里首先定义了文本宏 LINENUM, 它引用了 @LINE 汇编操作符, 该操作符用于返回当前源码行的行号:

```
LINENUM TEXTEQU %(@LINE)
```

接下来在包含 ECHO 语句的行中, 行首定义的展开操作符 (%) 导致 LINENUM 被展开:

```
%    ECHO *   Error on line LINENUM: EAX cannot be the second
```

假设下面的宏调用发生在程序的第 40 行:

```
MUL32 val1, eax, val3
```

在汇编过程中会显示以下信息:

```
-----
*   Error on line 40: EAX cannot be the second
*   argument when invoking the MUL32 macro.
-----
```

读者可以在 Macro3.asm 文件中查看测试 Mul32 宏的代码。

文本操作符 (<>)

文本操作符 (<>) 允许把多个字符或符号作为一个字符串整体进行传递, 以阻止预处理器把尖括号内的参数解释为单独的参数。该操作符在字符串包含逗号、百分号、连接符或分号的时候特别有用。如果不使用该操作符, 这些字符就会被解释成分隔符或其他操作符。例如, 本章前面给出的宏 mWrite 只接收一个唯一的字符串参数, 如果传递下面的字符串, 预处理器会将其解释为三个独立的参数:

```
mWrite "Line three", 0dh, 0ah
```

由于该宏只接收一个参数，所以第一个逗号后面的所有参数都将被忽略，如果把字符串用文本操作符括起来，预处理器就会把尖括号内所有文本作为一个宏参数进行传递：

```
mWrite <"Line three", 0dh, 0ah>
```

特殊文本字符操作符 (!)

特殊文本字符操作符(!)的目的和文本操作符的目的几乎是一样的：强制预处理器把预定义的操作符字符作为普通的字符。在下面的 TEXTTEQU 定义中，!操作符阻止了预处理器把">"符号解释为文本分隔符：

```
BadYValue TEXTTEQU <Warning: Y-coordinate is !> 24>
```

警告消息的例子：下面的例子有助于说明%、&和!操作符是如何一起使用的。假设前面已经定义了 BadYValue，下面创建的 ShowWarning 宏接收一个文本参数，把该文本参数用引号括起来并传递给 mWrite 宏，注意替换操作符(&)的用法：

```
ShowWarning MACRO message
    mWrite "&message"
ENDM
```

接下来我们调用 ShowWarning，把表达式%BadYValue 传递给它，%操作符对 BadYValue 求值并产生等价的字符串：

```
.code
ShowWarning %BadYValue
```

可能与预期的一样，程序运行并显示下面的警告信息：

```
Warning: Y-coordinate is >24
```

10.3.8 宏函数

宏函数与宏过程相似，它们都为汇编语句块赋予了一个名字，所不同的是，宏函数总是用 EXITM 伪指令返回一个常量（整数或字符串）。在下面的例子中，如果符号已定义则 IsDefined 宏返回真（-1），否则返回假（0）：

```
IsDefined MACRO symbol
    IFDEF symbol
        EXITM <-1>                ;; 真
    ELSE
        EXITM <0>                 ;; 假
    ENDIF
ENDM
```

EXITM（退出宏）指令停止了宏的进一步展开。

调用宏函数：调用宏函数时参数列表必须用圆括号括起来。例如，我们可以调用 IsDefined 宏，并向它传递一个已经或尚未定义的参数 RealMode：

```
IF IsDefined( RealMode )
    mov ax,@data
    mov ds,ax
ENDIF
```

如果编译器在编译该语句之前已经遇到了 RealMode 的定义，则编译下面两条语句：

```
mov ax,@data
mov ds,ax
```


同样的 IF 指令块也可以放在宏 Startup 内：

```
Startup MACRO
    IF IsDefined( RealMode )
        mov ax,@data
        mov ds,ax
    ENDIF
ENDM
```

像 IsDefined 这样的宏在设计那些要在多种不同的内存模式下汇编的程序时是非常有用的。例如可以用 IsDefined 来决定要包含哪个文件：

```
IF IsDefined( RealMode )
    INCLUDE Irvine16.inc
ELSE
    INCLUDE Irvine32.inc
ENDIF
```

定义 RealMode 符号：剩下的所有事情就是寻找一种定义 RealMode 符号的方法了。一种方法是在程序开始放上下面的语句：

```
RealMode = 1
```

此外，还可以使用编译器的命令行选项“-D”来定义符号，下面的 ML 命令行定义了 RealMode 符号并为其赋值 1：

```
ML -c -DRealMode=1 myProg.asm
```

相应地，编译保护模式程序的 ML 命令行不应该定义 RealMode 符号：

```
ML -c myProg.asm
```

HelloNew 程序：下面的程序 (HelloNew.asm) 使用了前面给出的宏在屏幕上显示一条消息：

```
TITLE Macro Functions                (HelloNew.asm)

INCLUDE Macros.inc
IF IsDefined( RealMode )
    INCLUDE Irvine16.inc
ELSE
    INCLUDE Irvine32.inc
ENDIF

.code
main PROC
    Startup
    mWrite <"This program can be assembled to run ",0dh,0ah>
    mWrite <"in both Real mode and Protected mode.",0dh,0ah>
    exit
main ENDP
END main
```

程序既可在实地址模式下编译，也可以在保护模式下编译。

10.3.9 本节习题

1. IFB 伪指令的用途是什么？
2. IFIDN 伪指令的用途是什么？
3. 哪条伪指令阻止宏的进一步展开？
4. IFIDNI 和 IFIDN 伪指令有什么不同？
5. IFDEF 伪指令的用途是什么？

6. 哪条伪指令标记条件汇编语句块的结束?
7. 给出一个有默认初始化值的宏参数的例子。
8. 列出所有可以在常量布尔表达式中使用的关系操作符。
9. 写一个使用 IF, ELSE 和 ENDIF 的小例子程序。
10. 写一条语句, 使用 IF 伪指令检查宏参数 Z 的值。如果 Z 小于 0, 则在汇编时显示一条消息表明 Z 是无效的。
11. 宏定义中 & 操作符的作用是什么?
12. 宏定义中 ! 操作符的作用是什么?
13. 宏定义中 % 操作符的作用是什么?
14. 写一个简短的宏, 解释说明宏参数嵌套在字符串内时 & 操作符的用法。
15. 假设有下面的 mLocate 宏的定义:

```
mLocate MACRO xval,yval
    IF xval LT 0                                ;; xval < 0?
        EXITM                                   ;; if so, exit
    ENDIF
    IF yval LT 0                                ;; yval < 0?
        EXITM                                   ;; if so, exit
    ENDIF
    mov bx,0                                    ;; video page 0
    mov ah,2                                    ;; locate cursor
    mov dh,yval
    mov dl,xval
    int 10h                                     ;; call the BIOS
ENDM
```

列出下面每条宏调用语句展开时预处理器生成的源代码:

```
.data
row BYTE 15
col BYTE 60
.code
mLocate -2,20
mLocate 10,20
mLocate col,row
```

10.4 定义重复块

MASM 中有一些用于生成重复语句块的循环伪指令: WHILE, REPEAT, FOR 和 FORC。与 LOOP 指令不同, 这些伪指令只能用汇编期间, 而且只能使用常量值作为循环条件和计数器:

- WHILE 伪指令根据一个布尔表达式来重复语句块。
- REPEAT 伪指令根据一个计数器来重复语句块。
- FOR 伪指令通过遍历一个符号列表中的每个符号来重复语句块。
- FORC 伪指令通过遍历一个字符串中的每个字符来重复语句块。

在例子程序 Repeat.asm 中有每个伪指令的解释说明。

10.4.1 WHILE 伪指令

只要特定的常量表达式为真, WHILE 伪指令就重复语句块。格式如下:

```

WHILE constExpression
    statements
ENDM

```

下面的代码显示了如何生成 1 到 F0000000h 之间的斐波那契数作为一系列编译时期的常量:

```

.data
val1 = 1
val2 = 1
DWORD val1 ; 前两个值
DWORD val2
val3 = val1 + val2
WHILE val3 LT 0F0000000h
    DWORD val3
    val1 = val2
    val2 = val3
    val3 = val1 + val2
ENDM

```

代码生成的常量值可在列表文件中 (.LST) 查看。

10.4.2 REPEAT 伪指令

REPEAT 伪指令以固定次数重复指令块。格式如下:

```

REPEAT constExpression
    statements
ENDM

```

constExpression 是一个无符号整数常量表达式, 它决定了重复的次数。REPEAT 伪指令可以像 DUP 伪指令一样用于定义数组。在下例中, 结构 WeatherReadings 包含了一个地理位置字符串, 后面跟一个降雨量 (rainfall) 和湿度 (humidity) 的数组:

```

WEEKS_PER_YEAR = 52
WeatherReadings STRUCT
    location BYTE 50 DUP(0)
    REPEAT WEEKS_PER_YEAR
        LOCAL rainfall, humidity
        rainfall DWORD ?
        humidity DWORD ?
    ENDM
WeatherReadings ENDS

```

其中的 LOCAL 伪指令用于避免在汇编时循环重复定义 rainfall 和 humidity 可能导致的重重复定义错误。

10.4.3 FOR 伪指令

FOR 伪指令通过遍历一个以逗号分隔的符号列表重复语句块, 符号列表中的每个符号都引发一次循环。格式如下所示:

```

FOR parameter, <arg1, arg2, arg3, ...>
    statements
ENDM

```

在第一次循环时, 参数 (*parameter*) 的取值为 *arg1*, 第二次循环时, 参数的取值为 *arg2*, 如此反复, 直到遍历完列表中的最后一个符号为止。

学生注册的例子: 现在我们来创建一个学生注册登记的方案, 其中使用 COURSE 结构来代表

课程, COURSE 结构中包含课程编号和学分数。另外使用 SEMESTER 结构表示一个学季, 其中包含了 6 个课程结构的数组和一个名为 NumCourses 的计数器:

```
COURSE STRUCT
    Number BYTE 9 DUP(?)
    Credits BYTE ?
COURSE ENDS

; A semester contains an array of courses.
SEMESTER STRUCT
    Courses COURSE 6 DUP(<>)
    NumCourses WORD ?
SEMESTER ENDS
```

我们可以使用 FOR 循环定义 4 个 SEMESTER 对象, 每个对象都有一个从尖括号中的符号列表中选出的不同名字:

```
.data
FOR semName, <Fall1999, Spring2000, Summer2000, Fall2000>
    semName SEMESTER <>
ENDM
```

如果查看列表文件, 就会发现如下的变量定义:

```
.data
Fall1999 SEMESTER <>
Spring2000 SEMESTER <>
Summer2000 SEMESTER <>
Fall2000 SEMESTER <>
```

10.4.4 FORC 伪指令

FORC 伪指令通过遍历一个字符串中的每个字符来重复语句块, 字符串中的每个字符都引发一次循环, 格式如下:

```
FORC parameter, <string>
    statements
ENDM
```

第一次循环开始时, 参数 (parameter) 等于字符串中的第一个字符; 第二次循环时, 参数等于字符串中的第二个字符。依次类推, 直到字符串结束。下例创建了一个字符查找列表, 表中包含了几个非字母字符。注意 < 和 > 前面必须加特殊文本字符操作符 (!), 以防止与 FORC 伪指令的格式参数冲突:

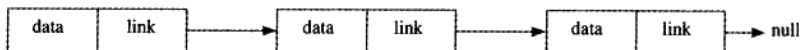
```
Delimiters LABEL BYTE
FORC code, <@#$$%^&*!<!>>
    BYTE "&code"
ENDM
```

在列表文件中可以看到, 该代码生成了下面的数据:

```
00000000 40 1 BYTE "@"
00000001 23 1 BYTE "#"
00000002 24 1 BYTE "$"
00000003 25 1 BYTE "%"
00000004 5E 1 BYTE "^"
00000005 26 1 BYTE "&"
00000006 2A 1 BYTE "*"
00000007 3C 1 BYTE "<"
00000008 3E 1 BYTE ">"
```

10.4.5 例子：链表

组合使用结构声明和 REPEAT 伪指令以使汇编器生成一个链表（Linked List）数据结构是相当简单的，链表中的每个节点包含一个数据（data）区和一个链接（link）区：



在每个节点的数据区内可以使用一个或多个变量存放节点的数据。在链接区中，用一个指针存放链表中下一个相邻节点的地址，最后一个节点的链接区通常包含一个空指针。下面编写一个创建和显示简单链表的程序。程序首先定义一个链表节点结构，里面包含一个整数（数据区）和一个指向下一个相邻节点的指针：

```

ListNode STRUCT
    NodeData DWORD ?           ; 节点的数据
    NextPtr  DWORD ?           ; 下一个节点的指针
ListNode ENDS
  
```

接下来使用 REPEAT 伪指令创建 ListNode 对象的多个实例。为了测试的目的，NodeData 域中包含了一个取值 1~15 的整数常量。在循环内部，我们增加计数器的值并把计数器的值插入到 ListNode 的各个域中：

```

TotalNodeCount = 15
NULL = 0
Counter = 0

.data
LinkedList LABEL PTR ListNode
REPEAT TotalNodeCount
    Counter = Counter + 1
    ListNode <Counter, ($ + Counter * SIZEOF ListNode)>
ENDM
  
```

表达式 $(\$ + \text{Counter} * \text{SIZEOF } \text{ListNode})$ 告诉编译器把当前计数器和 ListNode 的大小相乘并把乘积与当前地址计数器相加，然后再把结果插入到 ListNode 结构的 NextPtr 域中。注意：地址指针值中的 (\$) 总是固定为第一个节点的地址值，这很有趣。链表使用一个尾节点标记链表的结束，尾节点的 NextPtr 域为 0：

```
ListNode <0,0>
```

在程序遍历链表的时候，使用下面的语句获取 NextPtr 的值并同 NULL 相比较，以检查是否到达了链表的结尾：

```

mov  eax, (ListNode PTR [esi]).NextPtr
cmp  eax, NULL
  
```

程序清单：下面是程序的完整清单。在 main 过程中，程序使用循环遍历链表并显示所有节点的数据值，循环中没有使用固定的计数器，相反，程序通过检查 NULL 指针来判断是否到达了链表尾部，如果发现 NULL 指针则终止循环：

```

TITLE Creating a Linked List                                (List.asm)

INCLUDE Irvine32.inc

ListNode STRUCT
    NodeData DWORD ?
    NextPtr  DWORD ?
ListNode ENDS
  
```

```

TotalNodeCount = 15
NULL = 0
Counter = 0

.data
LinkedList LABEL PTR ListNode
REPEAT TotalNodeCount
    Counter = Counter + 1
    ListNode <Counter, ($ + Counter * SIZEOF ListNode)>
ENDM
ListNode <0,0> ; 尾节点

.code
main PROC
    mov     esi,OFFSET LinkedList
; 显示 NodeData 域中的整数
NextNode:
    ; 检查是否是尾节点
    mov     eax,(ListNode PTR [esi]).NextPtr
    cmp     eax,NULL
    je      quit
    ; 显示节点数据
    mov     eax,(ListNode PTR [esi]).NodeData
    call    WriteDec
    call    CrLf
    ; 获取下一个节点的指针
    mov     esi,(ListNode PTR [esi]).NextPtr
    jmp     NextNode

quit:
    exit
main ENDP
END main

```

10.4.6 本节习题

1. 简要描述 WHILE 伪指令。
2. 简要描述 REPEAT 伪指令。
3. 简要描述 FOR 伪指令。
4. 简要描述 FORC 伪指令。
5. 哪条循环伪指令最适用于生成字符查找表？
6. 写出下面的宏生成的语句：

```

FOR val,<100,20,30>
    BYTE 0,0,0,val
ENDM

```

7. 假设定义了下面的 mRepeat 宏：

```

mRepeat MACRO char,count
    LOCAL L1
    mov     cx,count
L1: mov     ah,2
    mov     dl,char
    int     21h
    loop    L1
ENDM

```

写出预处理器展开下面的语句时生成的代码:

```
mRepeat 'X',50          ; a
mRepeat AL,20            ; b
mRepeat byteVal,countVal ; c
```

8. 挑战: 在链表例子程序中 (10.4.5 节), 如果 REPEAT 循环编码如下, 结果会是什么?

```
REPEAT TotalNodeCount
    Counter = Counter + 1
    ListNode <Counter, ($ + SIZEOF ListNode)>
ENDM
```

10.5 本章小结

结构是在定义用户自定义类型时使用的模板或模式。MS-Windows 的 API 库中已经预定义了很多结构, 并用它们在应用程序和库之间传递数据。结构可以包含多种类型的域, 每个域声明都可以使用域初始值, 域初始值给了结构域一个默认值。

结构本身并不占用存储空间, 但结构变量占用存储空间, SIZEOF 操作符返回变量使用的字节数。

使用结构变量或类似于[esi] 的间接操作数加上点操作符可以引用结构的域, 在使用间接操作数来引用结构的域时, 必须使用 PTR 操作符来标识结构的类型, 如(COORD PTR [esi]).X。

结构中还可以包含结构域。Drunkard's Walk (醉汉走路) 程序中 (10.1.6 节) 演示了这样的例子, 其中的 DrunkardWalk 结构中又包含了 COORD 结构。

宏通常在程序的开始部分定义, 位于数据段和代码段之前。在后面调用宏时, 预处理器就在调用宏的地方插入宏包含的代码块的一份副本。

可用宏把过程调用封装起来, 以简化参数传递以及在栈上保存寄存器等工作。宏 mGotoxy, mDumpMem 和 mWriteString 调用了本书链接库中的过程, 它们都是使用宏封装过程调用例子。

宏过程 (或宏) 是命名的汇编语句块, 宏函数也是类似的, 它们之间的区别在于宏函数可以返回一个常量值。

IF, IFNB 和 IFIDNI 等条件汇编伪指令可以检查宏参数是否有超出范围、是否缺少参数或参数类型错误的情况。ECHO 伪指令在汇编过程中显示错误信息, 因此使得向程序员警告传递给宏的参数发生了错误成为可能。

替换操作符 (&) 能够解决对参数名的引用有歧义的情况; 展开操作符 (%) 展开文本宏并把常量表达式转化成文本; 文本操作符 (<>) 把多个字符和文本定义为一个字符串整体; 特殊文本字符操作符 (!) 强制预处理器把预定义的操作符作为普通字符对待。

重复块伪指令可以减少程序中重复代码的数量, 包括:

- WHILE 伪指令根据一个布尔表达式来重复语句块。
- REPEAT 伪指令根据一个计数器来重复语句块。
- FOR 伪指令通过遍历一个符号列表中的每个符号来重复语句块。
- FORC 伪指令通过遍历一个字符串中的每个字符来重复语句块。

10.6 编程练习

1. mReadKey 宏

创建一个宏, 等待击键动作并返回按下的键值, 宏应当包含两个分别用于存放按键的

ASCII 码和扫描码的参数。提示：调用本书链接库中的 `ReadKey` 过程。请编写一个程序测试该宏。例如下面的代码等待一个按键，当按键返回后两个参数中应该包含按键的 ASCII 码和扫描码：

```
.data
ascii BYTE ?
scan BYTE ?
.code
mReadkey ascii, scan
```

2. `mWriteStringAttr` 宏

（本习题要求阅读 15.3.3 节或 11.1.11 节。）编写一个宏，以指定的颜色在控制台上显示一个以空字符结尾的字符串，宏的参数应该包含字符串的名称和颜色。提示：调用本书链接库中的 `SetTextColor` 过程。写一个程序以几个不同颜色的字符串测试该宏。调用示例：

```
.data
myString db "Here is my string",0
.code
mWritestring myString, white
```

3. `mMove32` 宏

编写一个宏 `mMove32`，该宏的参数是两个 32 位内存操作数，宏应该把源操作数送到目的操作数中。写一个程序测试该宏。

4. `mMult32` 宏

编写宏 `mMult32`，把两个 32 位的内存操作数相乘并返回一个 32 位的乘积。写一个程序测试该宏。

5. `mReadInt` 宏

编写一个宏 `mReadInt`，从标准输入上读取一个 16 位或 32 位的有符号整数，并在给定的参数中返回。使用条件操作符以允许宏能够处理所期望大小的输入参数。写一个程序测试该宏并传递不同尺寸的操作数。

6. `mWriteInt` 宏

编写宏 `mWriteInt`，通过调用 `WriteInt` 库过程在标准输出上显示一个有符号整数。传递给宏的参数可以是一个字节、字或双字。在宏内使用条件操作符以便能适应不同尺寸的参数。写一个程序测试该宏，向宏传递不同尺寸的参数。

7. `mScroll` 宏

（要求首先阅读 15.3.3 节。）编写 `mScroll` 宏，在屏幕上显示彩色矩形。在宏定义中使用下面的参数，如果 `attrib` 为空，则使用黑色背景亮灰色字符的配色方案：

<code>ULrow</code>	窗口的左上角所在行
<code>ULcol</code>	窗口的左上角所在列
<code>LRrow</code>	窗口的右下角所在行
<code>LRcol</code>	窗口的右下角所在列
<code>attrib</code>	滚动行的色彩

写一个程序测试该宏。

8. Drunkard's Walk (醉汉走路)

在测试 Drunkard's Walk 例子程序时,读者可能已经注意到了教授漫步的时候并不会离开起点很远。毫无疑问,这是下面的事实所导致的:教授在向每个方向移动时有同样的概率。修改程序,以便教授在走完一步后有 50% 的可能性沿着原方向继续前进,向相反方向行进的概率是 10%,向左或向右转的概率各为 20%。在循环开始之前设置一个默认的起始方向。

9. 多双字移位

(使用 7.8 节的练习 4 的解答作为本习题的起点。)编写一个宏 mShiftDoubleWords, 把数组 arrayName 的内容向左或向右 (基于一个方向参数) 移动指定的数据位数:

```
mShiftDoublewords MACRO
    arrayName,           ;; 数组的名字
    direction,           ;; R 或 L
    numberOfBits          ;; 移位位数
```

10. 三操作数指令

一些计算机指令集允许三操作数的算术指令。这样的操作有时会出现在简单的虚拟汇编器中, 这些虚拟的汇编器是用于向学生介绍汇编语言概念或编译器的中间语言时使用的。在下面列出的宏中, 假设 EAX 保留用于宏的操作不需要在调用后恢复, 但其他宏修改的寄存器则需要恢复。所有的参数都是内存操作数, 编写宏模拟下面的操作:

```
a. add3 destination, source1, source2
b. sub3 destination, source1, source2 (destination = source1 - source2)
c. mul3 destination, source1, source2
d. div3 destination, source1, source2 (destination = source1 / source2)
```

例如, 下面的宏调用实现了表达式 $x = (w + y) \times z$:

```
.data
temp DWORD ?
.code
add3 temp, w, y           ; temp = w + y
mul3 x, temp, z           ; x = temp * z
```

写一个程序, 实现 4 个算术表达式以测试前面编写的宏, 每个表达式都应包含乘法操作。

第 11 章 MS-Windows 程序设计

本章要点

- Win32 控制台编程
- 编写 Windows 图形界面应用程序
- 动态内存分配
- IA-32 内存管理

11.1 Win32 控制台编程

在阅读本章的过程中，读者最好能随时提醒自己思考以下的问题：

- 32 位程序如何进行文本的输入和输出？
- 32 位控制台模式下如何处理文本颜色？
- Irvine32 库是如何工作的？
- Windows 下如何处理时间和日期？
- 如何使用 MS-Windows 函数读写数据文件？
- 使用汇编能写出图形界面的 Windows 应用程序吗？
- 在保护模式下，如何把段地址和偏移地址转换成物理地址？
- 为什么虚拟内存能带来诸多好处？

在讲述 MS-Windows 32 位编程基础知识的过程中，本章将回答上面这些（甚至于更多的）问题。在了解了一些数据结构和参数相关的知识之后，控制台程序是相当容易编写的，这就是为什么本章讨论的是 Win32 编程，而大部分的内容却是和文本模式的控制台程序相关的原因。Irvine32 链接库是完全基于 Win32 控制台函数实现的，因此读者可以把它的源代码和本章中的相关内容进行对比。Irvine32 链接库的源代码可在本书附带代码的 \Examples\Lib32 目录下找到。

为什么不从我们平常在 Windows 下看到的图形界面的窗口应用程序开始学习呢？最主要的原因是：这样会涉及繁多的技术细节，汇编语言或 C 语言编写的图形界面应用程序冗长而复杂。即使是一些优秀作者的帮助下，C 和 C++ 程序员们还是年复一年地为搞清楚图形设备句柄、消息传递、字体的度量、设备位图和映射模式等各种技术细节而感到吃力。事实上，在因特网上有很多汇编爱好者很擅长 Windows 图形界面编程，作者已经对他们的网站进行了链接，参见本书网站（www.asmirvine.com）上 Assembly Language Sources 部分的链接。

但是希望了解图形界面编程的读者也不必感到失望，11.2 节介绍了一些这方面的知识。虽然这仅仅只是个开始，但可能对读者以后深入这些主题会有所启发。在 11.5 节中，也列出了用于深入了解这方面内容的一些参考文献。

从表面看，32 位的控制台程序和 16 位的 MS-DOS 文本应用程序在外观和行为上都是很相似的。不过事实上，32 位控制台程序和 MS-DOS 程序还是有一些不同：前者在保护模式下运行，后者在实模式下运行。它们使用的是完全不同的函数库，Win32 控制台程序使用的就是 Windows 图形界面程序使用的那些库文件，而 MS-DOS 程序使用的是 BIOS 和 MS-DOS 中断，这些中断在出现 IBM-PC 的那个年代就已经在使用了。

应用程序编程接口（API，Application Programming Interface）是一些类型、常量和函数的集合，它提供了直接通过编程操纵对象的途径。

Win32 平台软件开发包（Win32 Platform SDK）：和 Win32 API 有紧密联系的是 Microsoft Platform SDK，SDK 的含义是软件开发包（Software Development Kit），这是一些用于创建 Windows 应用程序的工具软件、库文件、代码例子和帮助文档的集合。完整的文档也可以在 Microsoft 的网站上找到，可在 www.msdn.microsoft.com 上搜索“Platform SDK”。下载 Platform SDK 是免费的。

提示：Irvine32 链接库和 Win32 API 是兼容的，因此在程序中可以同时调用 Irvine32 库中的函数和 Win32 API 中的函数。

11.1.1 背景知识

当一个 Windows 应用程序开始运行的时候，它可以创建一个控制台窗口，也可以创建一个图形化的窗口。可以在项目文件中为 LINK 命令指定下面的命令行选项，该选项通知链接器创建基于控制台的应用程序：

```
/SUBSYSTEM:CONSOLE
```

接下来可以看到，控制台程序的外观和行为看起来就像一个增强版的 MS-DOS 窗口程序。每个控制台程序有一个输入缓冲区、一个或者多个屏幕缓冲区：

- 输入缓冲区中包含了一个输入记录队列，每个输入记录都包含了一个输入事件的相关数据。输入事件包括键盘输入、鼠标单击或者用户在改变控制台窗口等事件。
- 屏幕缓冲区是一个包含了字符和颜色数据的二维数组，这些数据控制着控制台窗口中显示的文本的外观。

Win32 API 的参考信息

函数：本章只能介绍一部分 Win32 API 函数并给出一些简单的示例，由于篇幅所限，很多细节不能一一涉及。要查阅更多相关的内容，可以在 Microsoft Visual C++ Express 内单击帮助菜单或在线访问 Microsoft MSDN 站点（当前的网址是 www.msdn.microsoft.com）。在搜索函数或标识符时，应把过滤条件参数（“Filtered by”）设为“Platform SDK”。在本书附带的例子代码中，kernel32.txt 和 user32.txt 两个文件分别列出了 kernel32.lib 和 user32.lib 两个库中全部函数的名字。

常量：在阅读 Win32 API 中的函数文档时，经常会遇到一些常量名，比如 TIME_ZONE_ID_UNKNOWN，这类常量有的已经在 SmallWin.inc 中定义了。如果在 SmallWin.inc 中找不到定义，那么可以查阅本书网站上最新版本的 SmallWin.inc 是否已经包含了该定义。如果依旧没有找到，那么可以参考 SDK 中的相关头文件。例如，头文件 WinNT.h 中定义了 TIME_ZONE_ID_UNKNOWN 以及其他一些相关的常量：

```
#define TIME_ZONE_ID_UNKNOWN 0
#define TIME_ZONE_ID_STANDARD 1
#define TIME_ZONE_ID_DAYLIGHT 2
```

使用类似上面的信息，就可以在 SmallWin.inc 或自己的头文件中增加常量定义了，如：

```
TIME_ZONE_ID_UNKNOWN = 0
TIME_ZONE_ID_STANDARD = 1
TIME_ZONE_ID_DAYLIGHT = 2
```

字符集和 Windows API 函数

Win32 API 函数使用两种字符集：8 位的 ASCII/ANSI 字符集和 16 位的 Unicode 字符集（在

Windows NT/2000/XP 中提供)。用于处理文本的 Win32 API 函数往往提供了两个不同的版本：一个版本的函数名是以 A 结尾的(用于 8 位 ANSI 字符集);另一个版本的函数名是以 W 结尾的(用于 16 位的宽字符集,包括 Unicode 字符集)。以 WriteConsole 函数为例,两个不同版本的函数名如下所示:

- WriteConsoleA
- WriteConsoleW

Windows 95/98 操作系统不支持以 W 结尾的函数名。在 Windows NT/2000/XP 操作系统中,Unicode 是内置的字符集,在这些系统中如果调用 WriteConsoleA 函数,操作系统首先把 ANSI 字符转换成 Unicode 字符,然后再去调用 WriteConsoleW 函数。

在 MSDN 文档中,函数名(如 WriteConsole)尾部的 A 或者 W 省略掉了。在本书例子程序使用的 include 文件中,使用下面的方式对函数名重新定义:

```
WriteConsole EQU <WriteConsoleA>
```

这样就可以通过正常的函数名来调用 WriteConsole 函数了。

高级操作和底层操作

对控制台可以有两种层次的操作,使用者可以在简易性和全面性之间折衷选择:

- 高级操作函数从输入缓冲区中读取字符流,输出字符则被写到屏幕缓冲区中。输入和输出都可以被重新定向到一个文本文件的读写操作中。
- 底层操作用来获取键盘和鼠标操作的详细信息,以及用户和控制台窗口的交互动作(如拖动窗口、改变窗口大小等)。通过底层操作,也可以对控制台窗口的位置和大小、窗口中的字符颜色进行控制。

Windows 的数据类型

在文档中,Windows API 函数的声明以 C/C++ 的语法形式出现,在这些声明中,所有函数的参数类型都是基于标准 C 的数据类型或者 Windows 的预定义类型的(如表 11.1 所示)。正确区分数据类型中的数据值和数据值指针是很重要的,这些数据类型中以 LP 开头的都是指向其他对象的长指针。

表 11.1 Windows 的数据类型与 MASM 数据类型对照表

MS-Windows 类型	MASM 类型	说 明
BOOL, BOOLEAN	DWORD	布尔值(TRUE 或 FALSE)
BYTE	BYTE	8 位无符号整数
CHAR	BYTE	8 位 Windows ANSI 字符
COLORREF	DWORD	32 位数值,表示颜色值
DWORD	DWORD	32 位无符号整数
HANDLE	DWORD	对象的句柄
HFILE	DWORD	OpenFile 打开的文件句柄
INT	SDWORD	32 位有符号整数
LONG	SDWORD	32 位有符号整数
LPARAM	DWORD	Windows 过程和回调函数使用的消息参数类型
LPCSTR	PTR BYTE	指向(空字符结尾的)常量字符串类型数据的 32 位指针

(续表)

MS-Windows 类型	MASM 类型	说 明
LPCVOID	DWORD	指向任何类型常量的指针
LPSTR	PTR BYTE	指向 (空字符结尾的) 字符串类型数据的 32 位指针
LPCTSTR	PTR WORD	对 Unicode 和双字节字符集可移植的常量字符串的 32 位指针
LPTSTR	PTR WORD	对 Unicode 和双字节字符集可移植的字符串的 32 位指针
LPVOID	DWORD	指向任何未知类型的 32 位指针
LRESULT	DWORD	Windows 过程或回调函数的 32 位返回值类型
SIZE_T	DWORD	指针能指向的最大字节数
UINT	DWORD	32 位无符号整数
WNDPROC	DWORD	指向 Windows 过程的 32 位指针
WORD	WORD	16 位无符号整数
WPARAM	DWORD	作为参数传递给 Windows 过程或回调函数的 32 位值类型

SmallWin.inc 包含文件

SmallWin.inc 文件是本书作者创建的, 其中包含了用于 Win32 API 编程的常量定义、文本宏和函数原型。本书例子代码自始至终在使用的 Irvine32.inc 就包含了该文件, 因此任何包含了 Irvine32.inc 的程序都自动包含了 SmallWin.inc。如果安装了本书附带的例子代码, 则可以在 Examples\Lib32 目录下找到该文件, 其中的大多数常量定义都可以在用于 C/C++ 编程的 SDK 头文件 Windows.h 中找到。与 SmallWin.inc 这个名字的含义截然相反, 该文件非常大, 因此这里只给出一些示例:

```
DO_NOT_SHARE = 0
NULL = 0
TRUE = 1
FALSE = 0

; Win32 Console handles
STD_INPUT_HANDLE EQU -10
STD_OUTPUT_HANDLE EQU -11
STD_ERROR_HANDLE EQU -12
```

HANDLE 类型实际上是 DWORD 类型的别名, 有助于使汇编语言的函数原型声明与 Microsoft Win32 文档中给出的尽量一致:

```
HANDLE TEXTEQU <DWORD>
```

SmallWin.inc 中也包含了 Win32 调用中使用的结构的定义, 下面是两个例子:

```
COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS

SYSTEMTIME STRUCT
    wYear WORD ?
    wMonth WORD ?
    wDayOfWeek WORD ?
    wDay WORD ?
    wHour WORD ?
    wMinute WORD ?
    wSecond WORD ?
    wMilliseconds WORD ?
SYSTEMTIME ENDS
```

最后要说明的是, SmallWin.inc 中包含了本章介绍的所有 Win32 函数的原型声明。

控制台句柄

几乎所有的控制台函数都要求把控制台句柄作为第一个参数传递给它们,句柄是一个 32 位的无符号整数,唯一地标识了一个对象:如位图、画笔或者某个输入输出设备等。在这里我们可以使用下列句柄:

STD_INPUT_HANDLE	标准输入句柄
STD_OUTPUT_HANDLE	标准输出句柄
STD_ERROR_HANDLE	标准错误输出句柄

后面两个句柄用于向当前活跃的屏幕缓冲区输出数据。

GetStdHandle 函数用于获取一个对应控制台输入、输出或者错误输出流的句柄,在控制台程序中进行任何的输入输出操作都需要用到这样一个句柄,这里是函数原型:

```
GetStdHandle PROTO,
    nStdHandle:HANDLE ; 句柄的类型
```

nStdHandle 参数可以是 STD_INPUT_HANDLE、STD_OUTPUT_HANDLE 或者 STD_ERROR_HANDLE。函数在 EAX 中返回句柄,应该把它复制到一个变量中保存起来。下面是调用示例:

```
.data
inputHandle DWORD ?
.code
    INVOKE GetStdHandle, STD_INPUT_HANDLE
    mov inputHandle,eax
```

11.1.2 Win32 控制台函数

表 11.2 是所有 Win32 控制台函数的速查列表^①,可以在 MSDN 网站(www.msdn.microsoft.com) 中找到这些函数的完整说明。

提示: Win32 API 函数不保护 EAX, EBX, ECX, EDX 寄存器,因此必须自己保护这些寄存器。

表 11.2 Win32 控制台函数

函 数	说 明
AllocConsole	为调用的进程分配一个新的控制台
CreateConsoleScreenBuffer	创建一个控制台屏幕缓冲区
ExitProcess	结束一个进程和它所有的线程
FillConsoleOutputAttribute	为指定数量的字符指定文本颜色和背景颜色
FillConsoleOutputCharacter	把一个字符输出到控制台中,重复指定的次数
FlushConsoleInputBuffer	刷新控制台输入缓冲区
FreeConsole	释放控制台
GenerateConsoleCtrlEvent	向某个共享进程所属的控制台的进程组发送控制信号
GetConsoleCP	获取某个进程所属的控制台输入时使用的代码页
GetConsoleCursorInfo	获取某个控制台屏幕缓冲区的光标属性,如大小、是否可见等
GetConsoleMode	获取某个控制台输入缓冲区的输入模式,或屏幕缓冲区的输出模式
GetConsoleOutputCP	获取某个进程所属的控制台输出时使用的代码页

① 来源: Microsoft MSDN 文档。

(续表)

函 数	说 明
GetConsoleScreenBufferInfo	获取某个控制台屏幕缓冲区的相关信息
GetConsoleTitle	获取控制台窗口的标题栏文字
GetConsoleWindow	获取某个进程所属控制台窗口的窗口句柄
GetLargestConsoleWindowSize	获取控制台窗口的最大可能尺寸
GetNumberOfConsoleInputEvents	获取控制台输入缓冲区中现存输入记录的数量
GetNumberOfConsoleMouseButtons	获取控制台使用的鼠标按键的数量
GetStdHandle	获取标准控制台输入、输出或者错误输出句柄
HandlerRoutine	供 SetConsoleCtrlHandler 函数使用的用户自定义函数
PeekConsoleInput	查看控制台输入缓冲区中的数据 (获取的数据不被清除)
ReadConsole	获取控制台输入缓冲区中的字符数据 (获取的数据被清除)
ReadConsoleInput	获取控制台输入缓冲区中的数据 (获取的数据被清除)
ReadConsoleOutput	获取控制台屏幕缓冲区中指定方块区域中的字符及其颜色属性
ReadConsoleOutputAttribute	获取控制台屏幕缓冲区中指定的连续位置的前景色和背景色
ReadConsoleOutputCharacter	获取控制台屏幕缓冲区中指定的连续位置的字符
ScrollConsoleScreenBuffer	移动屏幕缓冲区中的一块数据
SetConsoleActiveScreenBuffer	把指定的屏幕缓冲区设置为当前显示的屏幕缓冲区
SetConsoleCP	设置某个进程所属的控制台输入时使用的代码页
SetConsoleCtrlHandler	为某个进程挂上或者清除程序自定义的控制处理的回调子程序
SetConsoleCursorInfo	设置某个控制台屏幕缓冲区的光标属性, 如大小、是否可见等
SetConsoleCursorPosition	设置某个控制台屏幕缓冲区中的光标位置
SetConsoleMode	设置某个控制台输入缓冲区的输入模式, 或屏幕缓冲区的输出模式
SetConsoleOutputCP	设置某个进程所属的控制台输出时使用的代码页
SetConsoleScreenBufferSize	改变某个控制台屏幕缓冲区的大小
SetConsoleTextAttribute	设置写到屏幕缓冲区中的字符的颜色和背景色
SetConsoleTitle	设置控制台窗口标题栏上面的文字
SetConsoleWindowInfo	设置屏幕缓冲区窗口的大小和位置
SetStdHandle	设置控制台标准输入、输出和错误输出句柄
WriteConsole	在控制台屏幕缓冲区的当前光标位置写入一个字符串
WriteConsoleInput	把数据直接写入控制台输入缓冲区
WriteConsoleOutput	在控制台屏幕缓冲区的矩形区域中写入字符及其颜色属性
WriteConsoleOutputAttribute	设置控制台屏幕缓冲区中连续位置的前景色和背景色
WriteConsoleOutputCharacter	在控制台屏幕缓冲区中的连续位置中写入字符

11.1.3 显示消息框

在 Win32 程序中产生输出的最简单方法之一就是调用 MessageBoxA 函数:

```

MessageBoxA PROTO,
    hWnd:DWORD,           ; 窗口句柄 (可以为空)
    lpText:PTR BYTE,      ; 消息框内的字符串
    lpCaption:PTR BYTE,   ; 对话框标题字符串
    uType:DWORD           ; 内容和行为类型

```

在基于控制台的应用程序中，可以把 `hWnd` 设为 `NULL`，以表示消息框没有所有者；`lpText` 参数是要显示在消息框内的（空字符结尾的）字符串的指针；`lpCaption` 参数是要显示的对话框标题字符串（空字符结尾的）的指针；`uType` 参数指定对话框的内容和行为。

对话框的内容和行为：`uType` 参数是一个位映射整数，包含了三类选项：要显示的按钮、图标以及默认的按钮。可显示按钮的可能组合值如下：

- `MB_OK`
- `MB_OKCANCEL`
- `MB_YESNO`
- `MB_YESNOCANCEL`
- `MB_RETRYCANCEL`
- `MB_ABORTRETRYIGNORE`
- `MB_CANCELTRYCONTINUE`

默认按钮：可以指定在用户按下 `Enter` 键时自动选择哪个按钮。可用的选项包括 `MB_DEFBUTTON1`（默认），`MB_DEFBUTTON2`，`MB_DEFBUTTON3` 和 `MB_DEFBUTTON4`。按钮是从左到右计数的，第一个按钮的计数是 1。

图标：有 4 类图标可以选用，有时候多个常量表示同样的图标：

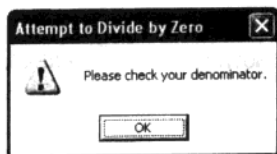
- 停止标志：`MB_ICONSTOP`，`MB_ICONHAND` 和 `MB_ICONERROR`。
- 问号（？）：`MB_ICONQUESTION`。
- 信息（i）：`MB_ICONINFORMATION`，`MB_ICONASTERISK`。
- 惊叹号（!）：`MB_ICONEXCLAMATION`，`MB_ICONWARNING`。

返回值：如果 `MessageBoxA` 失败，返回 0，否则返回一个整数，指明在关闭对话框时用户点击了哪个按钮，相关的常量包括：`IDABORT`，`IDCANCEL`，`IDCONTINUE`，`IDIGNORE`，`IDNO`，`IDOK`，`IDRETRY`，`IDTRYAGAIN` 和 `IDYES`。所有这些常量都在 `SmallWin.inc` 中定义了。

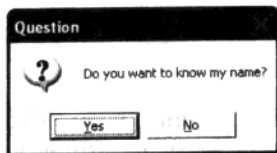
`SmallWin.inc` 把 `MessageBoxA` 重新定义为 `MessageBox`，后者看起来对用户更友好一些。

演示程序

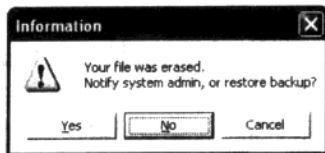
下面的程序（`MessageBox.asm`）演示了 `MessageBoxA` 函数的一些功能。第一个函数调用显示一条警告信息：



第二个函数调用询问一个问题，如果用户选择了 `Yes` 按钮，程序根据返回值显示一个字符串：



第三个按钮显示了三个按钮，问了一个看似毫无意义的问题：



程序清单：下面是程序的清单，由于 MessageBox 是 MessageBoxA 的别名，因此程序中使用了前者：

```

TITLE Demonstrate MessageBox          (MessageBox.asm)
INCLUDE Irvine32.inc

.data
captionW    BYTE "Attempt to Divide by Zero",0
warningMsg  BYTE "Please check your denominator.",0
captionQ    BYTE "Question",0
questionMsg BYTE "Do you want to know my name?",0
showMyName  BYTE "My name is MASM",0dh,0ah,0

captionC    BYTE "Information",0
infoMsg     BYTE "Your file was erased.",0dh,0ah
            BYTE "Notify system admin, or restore backup?",0

.code
main PROC
; 显示一条警告信息
    INVOKE MessageBox, NULL, ADDR warningMsg,
        ADDR captionW,
        MB_OK + MB_ICONEXCLAMATION

; 询问一个问题，等待回应
    INVOKE MessageBox, NULL, ADDR questionMsg,
        ADDR captionQ, MB_YESNO + MB_ICONQUESTION

    cmp     eax, IDYES                ; 单击了 YES 按钮?
    jne     L2                       ; 如果没有，则跳走

; 向控制台窗口写名称
    mov     edx, OFFSET showMyName
    call    WriteString

L2:

; 更复杂的按钮，可能会让用户迷惑
    INVOKE MessageBox, NULL, ADDR infoMsg,
        ADDR captionC, MB_YESNOCANCEL + MB_ICONEXCLAMATION \
            + MB_DEFBUTTON2

    exit
main ENDP
END main

```

如果想让对话框位于桌面上所有其他窗口之上，那么可以在最后一个参数 `uType` 中多传递一个（或操作）`MB_SYSTEMMODEL` 选项。

11.1.4 控制台输入

到现在为止，我们已经数次用到 `ReadString` 和 `ReadChar` 过程，这两个过程是由本书附带的链接库提供的，它们的使用相当简单，这样读者就可以把注意力集中在其他问题上了。这两个过程都是对 Win32 API 函数 `ReadConsole` 的封装（封装过程隐藏了被封装过程中的一些细节）。

控制台输入缓冲区：Win32 控制台有一个输入缓冲区，其中包含一个输入动作记录的队列，

每个输入动作，如键盘敲击、鼠标移动、按下鼠标键等，都会在缓冲区中产生一条记录。高级操作函数如 ReadConsole 等过滤并处理这些输入数据，只返回字符流。

ReadConsole 函数

ReadConsole 函数提供了一种把文本输入读取到一个缓冲区中的便捷方法，函数原型如下：

```
ReadConsole Proto,
    hConsoleInput:HANDLE,           ; 输入句柄
    lpBuffer:PTR BYTE,             ; 缓冲区地址指针
    nNumberOfCharsToRead:DWORD,    ; 要读取的字符数量
    lpNumberOfCharsRead:PTR DWORD,  ; 指向返回实际读取数量大小的指针
    lpReserved:DWORD               ; (保留)
```

hConsoleInput 参数是一个由 GetStdHandle 函数返回的有效输入句柄；lpBuffer 参数指向一个字符缓冲区；nNumberOfCharsToRead 参数是一个 32 位的整数，指定了要读取字符的最大数量；lpNumberOfCharsRead 参数是指向一个双字变量的指针，函数运行时会填写该变量，它返回实际读取到缓冲区中的字符数量。最后一个参数未使用，使用时要传递一个数值（比如 0）。

除了用户的输入以外，调用 ReadConsole 读入输入缓冲区中的文本还包含两个额外的字符——行结束字符（回车和换行符）。欲使输入缓冲区中的文本以 0 结尾，那么应该把包含 0Dh（回车）的字节替换为 0，ReadString 过程就是这样做的。

注：Win32 API 函数不保留 EAX, EBX, ECX 和 EDX 寄存器。

例子程序：设想一下，我们要写一个程序来读取用户输入的字符。首先，需要调用 GetStdHandle 函数获取控制台的标准输入句柄，然后使用这个句柄调用 ReadConsole 函数。下面的程序 ReadConsole.asm 演示了这项技术。注意，Win32 API 函数调用和对 Irvine32 库过程的调用是可以共存的，因此，下面的代码在调用 Win32 API 的同时还调用了 DumpMem 过程：

```
TITLE Read From the Console          (ReadConsole.asm)

INCLUDE Irvine32.inc
BufSize = 80

.data
buffer BYTE BufSize DUP(0),0,0
stdInHandle HANDLE ?
bytesRead DWORD ?

.code
main PROC
    ; 获取标准输入的句柄
    INVOKE GetStdHandle, STD_INPUT_HANDLE
    mov     stdInHandle,eax

    ; 等待用户输入
    INVOKE ReadConsole, stdInHandle, ADDR buffer,
        BufSize - 2, ADDR bytesRead, 0

    ; 显示缓冲区的内容
    mov     esi,OFFSET buffer
    mov     ecx,bytesRead
    mov     ebx,TYPE buffer
    call    DumpMem

    exit
main ENDP
END main
```

如果用户输入了“abcdefg”，程序将产生下面的输出。输入缓冲区中包含了 9 个字符：“abcdefg”以及 0Dh 和 0Ah，行结束符（0Dh, 0Ah）是用户按下回车键时输入缓冲区的。在这个例子中，bytesRead 等于 9。

```
Dump of offset 00404000
```

```
-----
61 62 63 64 65 66 67 0D 0A
```

错误的检查

如果 Windows API 函数返回了一个出错值 (如 NULL), 可以调用 GetLastError API 函数获取关于该错误的更多信息。GetLastError 在 EAX 中返回一个 32 位的错误码 (一个整数):

```
.data
messageId DWORD ?
.code
call GetLastError
mov messageId, eax
```

MS-Windows 有成千上万个错误码, 一个人不可能记住所有错误码的含义, 因此这时获取能够描述该错误码含义的字符串就非常有意义了, 这可以通过调用 FormatMessage 函数做到:

```
FormatMessage Proto,                ; 格式化一条消息
    dwFlags:DWORD,                  ; 格式化选项
    lpSource:DWORD,                 ; 消息定义的位置
    dwMsgID:DWORD,                  ; 消息 ID
    dwLanguageID:DWORD,             ; 语言 ID
    lpBuffer:PTR BYTE,              ; 指向接收字符串缓冲区的指针
    nSize:DWORD,                    ; 缓冲区的大小
    va_list:DWORD                   ; 参数列表的指针
```

参数有些复杂, 因此必须阅读 SDK 文档, 以获取其全貌。以下是最有用的值的简要描述。除了 lpBuffer 为输出参数外, 其余均为输入参数。

- dwFlags, 一个双字整数, 用于存放格式化选项, 如应该如何解释 lpSource 参数。这个参数指定了应如何处理换行以及格式化后的输出行的最大宽度, 推荐的选项值是 FORMAT_MESSAGE_ALLOCATE_BUFFER 和 FORMAT_MESSAGE_FROM_SYSTEM。
- lpSource, 指向消息定义的位置的指针, 对于上面推荐的 dwFlags 选项值, 应把该值设为 NULL (0)。
- dwMsgID, 调用 GetLastError 返回的双字整数值。
- dwLanguageID, 语言标识符。如果该值为 0, 则消息的语言是中性的, 也就是说消息的语言是用户的默认本地语言。
- lpBuffer, 指向接收消息字符串缓冲区的指针, 这是一个输出参数。如果 dwFlags 指定了 FORMAT_MESSAGE_ALLOCATE_BUFFER 选项, 输出缓冲区将自动分配, 这个参数也就无须指定了。
- nSize, 指定用于存放消息字符串的 lpBuffer 指向的缓冲区的大小。如果 dwFlags 指定了 FORMAT_MESSAGE_ALLOCATE_BUFFER 选项, 则该参数可以传递 0。
- va_list, 要插入到格式化后消息内的值的列表。由于我们通常不格式化出错消息, 因此该参数可设为 0。

下面是 FormatMessage 函数的调用示例:

```
.data
messageId DWORD ?
pErrorMsg DWORD ?                ; 指向错误消息
.code
call GetLastError
mov messageId, eax
INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
```

```
FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageID, 0,
ADDR pErrorMsg, 0, NULL
```

如果 dwFlags 指定了 FORMAT_MESSAGE_ALLOCATE_BUFFER 选项, 那么在调用 FormatMessage 函数之后, 还要调用 LocalFree 释放 FormatMessage 分配的内存:

```
INVOKE LocalFree, pErrorMsg
```

WriteWindowsMsg 过程: 本书链接库中包含了下面的 WriteWindowsMsg 过程, 封装了 API 函数调用的细节:

```
;-----
WriteWindowsMsg PROC USES eax edx
;
; Displays a string containing the most recent error
; generated by MS-Windows.
; Receives: nothing
; Returns: nothing
;-----
.data
WriteWindowsMsg_1 BYTE "Error ",0
WriteWindowsMsg_2 BYTE ": ";0
pErrorMsg DWORD ?           ; 指向错误消息
messageId DWORD ?
.code
    call    GetLastError
    mov     messageId,eax

; 显示出错码数字
    mov     edx,OFFSET WriteWindowsMsg_1
    call    WriteString
    call    WriteDec
    mov     edx,OFFSET WriteWindowsMsg_2
    call    WriteString

; 获取对应的消息字符串
    INVOKE  FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
        FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageId, NULL,
        ADDR pErrorMsg, NULL, NULL

; 显示 MS-Windows 产生的错误消息
    mov     edx,pErrorMsg
    call    WriteString

; 释放错误消息字符串占用的内存
    INVOKE  LocalFree, pErrorMsg

    ret
WriteWindowsMsg ENDP
```

单字符的输入

在控制台下进行单字符输入需要一点技巧。MS-Windows 为当前安装的键盘提供了一个设备驱动程序, 在按键按下的时候, 一个 8 位的扫描码被送到键盘端口; 在按键释放时, 又有一个扫描码被送到键盘端口。MS-Windows 使用设备驱动把扫描码转换成 16 位的虚拟键码 (virtual-key code)。虚拟键码是 MS-Windows 定义的设备相关的值, 用于标识按键的功用。MS-Windows 将创建一个包含了按键的扫描码、虚拟键码以及其他相关信息的消息, 然后把这个消息放在 Windows 消息队列中, 最后通过某种方式送达当前正在执行程序的线程 (这里用控制台输入句柄标识)。如果想了解键盘输入过程方面的更多信息, 请阅读 Platform SDK 文档中的主题 “About Keyboard Input”。要想查看虚拟键码常量值列表, 可参看本书附带代码\Examples\ch11 目录下的 VirtualKey.inc 文件。

Irvine32 库中键盘相关的过程: Irvine32 库中有两个与键盘相关的过程:

- ReadChar 等待键盘输入一个 ASCII 字符并在 AL 中返回该字符。
- ReadKey 过程检查键盘输入, 但不等待。如果键盘输入缓冲区中没有按键, 零标志置位; 如果输入缓冲区中有按键, 则零标志清零并且在 AL 中返回 0 或按键的 ASCII 码。EAX 和 EDX 的高半部分会被改写。

ReadKey 过程返回时, 如果 AL 中返回的是 0, 则表明按下的是一个特殊的键 (功能键、光标键等), AH 寄存器中包含了按键的扫描码, 在本书的前言部分可以找到特殊按键的扫描码表。DX 中包含了虚拟键码, EBX 包含了键盘控制键的状态信息。在调用 ReadKey 后, 可以使用 TEST 指令检查各个特殊键的值。ReadKey 的实现有点冗长, 这里就不再重复给出其代码了, 感兴趣的读者可自行查阅 Examples\Lib32 目录下的 Irvine32.asm 文件中的相应代码。键盘控制键的状态值如表 11.3 所示。

表 11.3 键盘控制键的状态值

值	含 义
CAPSLOCK_ON	Caps Lock 键盘灯亮
ENHANCED_KEY	按键是增强按键
LEFT_ALT_PRESSED	左 Alt 按下
LEFT_CTRL_PRESSED	左 Ctrl 按下
NUMLOCK_ON	Num Lock 键盘灯亮
RIGHT_ALT_PRESSED	右 Alt 按下
RIGHT_CTRL_PRESSED	右 Ctrl 按下
SCROLLLOCK_ON	Scroll Lock 键盘灯亮
SHIFT_PRESSED	Shift 按下

ReadKey 测试程序: 下面的程序测试 ReadKey, 程序使用一个循环延时等待按键, 然后报告是否按下了 CapsLock 键。正如第 5 章所提到的, 应该延时以留给 MS-Windows 处理消息循环的时间。

```

TITLE Testing ReadKey                (TestReadkey.asm)
INCLUDE Irvine32.inc
INCLUDE Macros.inc

.code
main PROC
L1:  mov     eax,10                    ; 为消息处理延时
     call    Delay
     call    ReadKey                  ; 等待按键
     jz      L1

     test    ebx,CAPSLOCK_ON
     jz      L2
     mWrite <"CapsLock is ON",0dh,0ah>
     jmp     L3
L2:  mWrite <"CapsLock is OFF",0dh,0ah>
L3:  exit
main ENDP
END main

```

获取键盘状态

调用 GetKeyState API 函数可以测试单个按键的状态, 查看其是否正被按下。函数原型如下。

GetKeyState PROTO, nVirtKey:DWORD

调用 GetKeyState 时, 应传递一个要检查按键的虚拟键码, 在函数返回后, 应测试 EAX 中的

相应位是否置位了。一些虚拟键码值以及应检查的对应数据位如表 11.4 所示。

表 11.4 使用 GetKeyState 测试按键

按 键	虚拟键码符号	应测试的 EAX 中的数据位
NumLock	VK_NUMLOCK	0
Scroll Lock	VK_SCROLL	0
左 shift	VK_LSHIFT	15
右 Shift	VK_RSHIFT	15
左 Ctrl	VK_LCONTROL	15
右 Ctrl	VK_RCONTROL	15
左 Menu	VK_LMENU	15
右 Menu	VK_RMENU	15

下面的例子程序演示了 GetKeyState 的用法，程序检查了 NumLock 和左 Shift 按键的状态：

```

TITLE Keyboard Toggle Keys          (Keybd.asm)
INCLUDE Irvine32.inc
INCLUDE Macros.inc

; GetKeyState sets bit 0 in EAX if a toggle key is
; currently on (CapsLock, NumLock, ScrollLock).
; Sets bit 15 in EAX if another specified key is
; currently down.

.code
main PROC

    INVOKE GetKeyState, VK_NUMLOCK
    test al,1
    .IF !Zero?
        mWrite <"The NumLock key is ON",0dh,0ah>
    .ENDIF

    INVOKE GetKeyState, VK_LSHIFT
    test al,80h
    .IF !Zero?
        mWrite <"The Left Shift key is currently DOWN",0dh,0ah>
    .ENDIF

    exit
main ENDP
END main

```

11.1.5 控制台输出

在前面的章节中，我们尽量使控制台输出尽可能简单，第 5 章中介绍的 Irvine32 链接库中的 WriteString 过程只要求一个参数：通过 EDX 传递的字符串地址。事实上，WriteString 过程是对 Win32 函数 WriteConsole 的封装，调用后者时处理的细节要更多一些。

不过本节还是要讲述如何直接调用 WriteConsole 和 WriteConsoleOutputCharacter 等 Win32 函数，直接调用这些函数需要了解更多的细节，但比使用 Irvine32 中的过程灵活性更大。

相关的数据结构

一些 Win32 控制台函数使用预定义的数据结构，如 COORD 和 SMALL_RECT 结构。COORD 结构用于存放字符在控制台屏幕缓冲区中的坐标，坐标系的原点(0,0)在屏幕的左上角：

```

COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS

```

SMALL_RECT 结构用于存放矩形区域的左上角和右下角坐标,它指定了控制台窗口中的一块矩形区域:

```
SMALL_RECT STRUCT
    Left   WORD ?
    Top    WORD ?
    Right  WORD ?
    Bottom WORD ?
SMALL_RECT ENDS
```

WriteConsole 函数

WriteConsole 函数在控制台窗口中的当前光标位置显示一个字符串并前进光标,支持标准的 ASCII 控制字符,如制表符、回车、换行符等。要显示的字符串不必以 0 结尾。函数原型如下:

```
WriteConsole PROTO,
    hConsoleOutput:HANDLE,
    lpBuffer:PTR BYTE,
    nNumberOfCharsToWrite:DWORD,
    lpNumberOfCharsWritten:PTR DWORD,
    lpReserved:DWORD
```

第一个参数 hConsoleOutput 是控制台输出的句柄;第二个参数 lpBuffer 是指向要显示的字符串的指针;第三个参数 nNumberOfCharsToWrite 指定了要显示的字符串的长度;第四个参数 lpNumberOfCharsWritten 指向一个整数变量,函数通过该变量返回实际输出的字符数量;最后一个参数是保留未用的,在使用的时候把它设置为 0。

例子程序: Console1

下面的 Console1.asm 程序在控制台窗口中显示一个字符串,以此示范了 GetStdHandle、ExitProcess 和 WriteConsole 函数的用法:

```
TITLE Win32 Console Example #1                                (Console1.asm)

; This program calls the following Win32 Console functions:
; GetStdHandle, ExitProcess, WriteConsole

INCLUDE Irvine32.inc

.data
endl EQU <0dh,0ah>                                           ; 行结束符
message LABEL BYTE
    BYTE "This program is a simple demonstration of"
    BYTE "console mode output, using the GetStdHandle"
    BYTE "and WriteConsole functions.",endl
messageSize DWORD ($-message)

consoleHandle HANDLE 0                                         ; 标准输出设备的句柄
bytesWritten  DWORD ?                                         ; 已输出的字符数量

.code
main PROC
    ; 获取控制台输出的句柄
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    mov consoleHandle,eax

    ; 在控制台上显示一个字符串
    INVOKE WriteConsole,
        consoleHandle,                                         ; 控制台输出句柄
        ADDR message,                                          ; 字符串的指针
        messageSize,                                           ; 字符串的长度
        ADDR bytesWritten,                                     ; 返回已输出的字节数
        0                                                       ; 未用

    INVOKE ExitProcess,0
main ENDP
END main
```

程序输出的内容如下所示：

```
This program is a simple demonstration of console mode output, using the
GetStdHandle and WriteConsole functions.
```

WriteConsoleOutputCharacter 函数

WriteConsoleOutputCharacter 函数将一定数量的字符复制到屏幕缓冲区从指定位置开始的连续空间中。函数原型如下所示：

```
WriteConsoleOutputCharacter PROTO,
    hConsoleOutput:HANDLE,           ; 控制台输出句柄
    lpCharacter:PTR BYTE,             ; 字符缓冲区的地址
    nLength:DWORD,                   ; 缓冲区的大小
    dwWriteCoord:COORD,              ; 首字符的坐标
    lpNumberOfCharsWritten:PTR DWORD ; 实际输出字符的数量
```

输出字符的时候，如果到达了屏幕行的末尾，那么自动换行。该函数不影响控制台缓冲区中原有字符的属性值。如果函数无法输出字符，返回值为 0，函数忽略字符串中的 ASCII 控制字符，如制表符、回车符和换行符。

11.1.6 文件的读写

CreateFile 函数

CreateFile 函数既可以用于创建一个文件，也可以用于打开一个文件。如果函数执行成功，那么它返回文件句柄，否则返回的是 INVALID_HANDLE_VALUE 常量。函数原型如下所示：

```
CreateFile PROTO,           ; 创建新文件或打开已存在的文件
    lpFilename:PTR BYTE,    ; 文件名字符串指针
    dwDesiredAccess:DWORD,   ; 存取模式
    dwShareMode:DWORD,       ; 共享模式
    lpSecurityAttributes:DWORD, ; 指向安全属性结构
    dwCreationDisposition:DWORD, ; 选项
    dwFlagsAndAttributes:DWORD, ; 文件属性
    hTemplateFile:DWORD      ; 模板文件的句柄
```

函数的参数如表 11.5 所示。

表 11.5 CreateFile 函数的参数

参 数	说 明
lpFilename	指向以 0 结尾的文件名字符串，文件名既可以是带部分路径的，也可以是全路径的文件名（即驱动器名\路径\文件名）
dwDesiredAccess	指定了如何访问文件（读或写文件）
dwShareMode	控制在文件打开后是否允许多个程序同时访问这个文件
lpSecurityAttributes	指向控制安全权限的一个结构
dwCreationDisposition	参数指明当文件存在或者不存在时要采取何种动作
dwFlagsAndAttributes	包含一系列的标志位，用来指定文件的各种属性，如归档属性、加密属性、隐含属性、系统属性、普通属性或者临时文件等
hTemplateFile	指定一个文件句柄，函数根据这个文件的属性和扩展属性设置当前新建文件的属性。如果不使用这个参数，那么只要把它设置为 0 就可以了

dwDesiredAccess：通过设置 dwDesiredAccess 参数，可以选择读模式、写模式、读写模式或者设备查询模式。可以同时选择表 11.6 列出的各种模式，当然还可以同时再加上很多未列在表中的标志位（在 Platform SDK 文档中搜索 CreateFile）：

表 11.6 dwDesiredAccess 参数选项

取 值	含 义
0	指定对象的设备查询模式。应用程序无须访问设备就可以查询它的属性，可以用来检查文件是否存在
GENERIC_READ	对一个对象进行读操作。从文件读取的话，文件读写指针可同时移动。如果要指定为读写模式，需要和下面的 GENERIC_WRITE 标志同时使用
GENERIC_WRITE	对一个对象进行写操作，对文件写入的时候，文件读写指针可同时移动。如果要指定为读写模式，需要和上面的 GENERIC_READ 标志同时使用

dwCreationDisposition：当文件已经存在或者不存在时，dwCreationDisposition 参数指定了要采取何种动作，参数必须指定为表 11.7 中所列的一种。

表 11.7 dwCreationDisposition 参数选项

取 值	含 义
CREATE_NEW	创建一个新文件。如果文件已经存在，函数执行失败
CREATE_ALWAYS	创建一个新文件。如果文件已经存在，那么原有文件被覆盖，同时清除原有的文件属性，新的属性值是函数参数指定的属性加上 FILE_ATTRIBUTE_ARCHIVE 属性，要求 dwDesiredAccess 参数包含 GENERIC_WRITE 属性值
OPEN_EXISTING	打开一个现存的文件。如果文件不存在，那么函数执行失败。可对打开的文件执行读写操作
OPEN_ALWAYS	打开一个现存的文件。如果文件不存在，那么函数使用 CREATE_NEW 方式创建文件
TRUNCATE_EXISTING	打开一个现存的文件。当文件打开后，函数自动把它的内容清除并把文件长度设置为 0。使用这个标志的时候，dwDesiredAccess 中必须至少指定 GENERIC_WRITE 标志。如果文件不存在，函数执行失败

表 11.8 列出了 dwFlagsAndAttributes 参数最常用的取值（完整的取值列表可在 Platform SDK 文档中查询 CreateFile）。这些取值可以组合起来使用，只不过任何其他的标志都会覆盖掉 FILE_ATTRIBUTE_NORMAL 属性。这些属性值都是 2 的幂值，因此可以使用汇编时的 OR 操作符或 + 操作符把多个标志组合成一个参数：

```
FILE_ATTRIBUTE_HIDDEN OR FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN+FILE_ATTRIBUTE_READONLY
```

表 11.8 dwFlagsAndAttributes 参数的取值

属 性	含 义
FILE_ATTRIBUTE_ARCHIVE	归档标志。用于备份和删除
FILE_ATTRIBUTE_HIDDEN	隐含标志。在（普通）列目录的时候文件不被列出
FILE_ATTRIBUTE_NORMAL	没有任何其他属性，这个标志必须单独使用
FILE_ATTRIBUTE_READONLY	只读属性。应用程序可以读取文件，但是不能进行写或者删除操作
FILE_ATTRIBUTE_TEMPORARY	文件作为临时存储用

例子：以下的例子演示了如何创建或者打开文件，但仅仅是说明性的，更多选项的使用方法请参考 MSDN 文档中的 CreateFile 部分。

- 打开一个现存的文件进行读操作（输入）：

```

INVOKE CreateFile,
    ADDR filename,                ; 文件名指针
    GENERIC_READ,                 ; 读取模式
    DO_NOT_SHARE,                 ; 共享模式为不共享
    NULL,                         ; 安全属性的指针
    OPEN_EXISTING,                ; 打开已存在的文件
    FILE_ATTRIBUTE_NORMAL,        ; 普通文件属性
    0                             ; 未用

```

- 打开一个现存的文件进行写操作（输出）。文件打开之后，可以覆盖已存在的数据，也可以把文件指针移动到文件的末尾，追加新的数据（参见 SetFilePointer 函数，11.1.6 节）：

```

INVOKE CreateFile,
    ADDR filename,
    GENERIC_WRITE,                 ; 写入文件
    DO_NOT_SHARE,
    NULL,
    OPEN_EXISTING,                 ; 文件必须存在
    FILE_ATTRIBUTE_NORMAL,
    0

```

- 创建一个普通属性的新文件，如果文件存在则覆盖：

```

INVOKE CreateFile,
    ADDR filename,
    GENERIC_WRITE,                 ; 写入文件
    DO_NOT_SHARE,
    NULL,
    CREATE_ALWAYS,                 ; 覆盖现存的文件
    FILE_ATTRIBUTE_NORMAL,
    0

```

- 如果文件不存在则创建一个新文件，否则就打开现存的文件进行输出：

```

INVOKE CreateFile,
    ADDR filename,
    GENERIC_WRITE,                 ; 写入文件
    DO_NOT_SHARE,
    NULL,
    CREATE_NEW,                     ; 不会删除现存文件
    FILE_ATTRIBUTE_NORMAL,
    0

```

（常量 DO_NOT_SHARE 和 NULL 已经在 SmallWin.inc 文件中定义了，SmallWin.inc 被 Irvine32.inc 文件自动包含了。）

CloseHandle 函数

CloseHandle 函数关闭一个已经打开对象的句柄。函数原型如下所示：

```

CloseHandle PROTO,
    hObject:HANDLE                ; 对象句柄

```

可利用 CloseHandle 来关闭当前打开文件的名柄，如果失败则返回 0。

ReadFile 函数

ReadFile 函数从一个输入文件中读取数据。函数原型如下所示：

```

ReadFile PROTO,
    hFile:HANDLE,                ; 输入文件句柄
    lpBuffer:PTR BYTE,           ; 缓冲区指针
    nNumberOfBytesToRead:DWORD,   ; 要读取的字节数
    lpNumberOfBytesRead:PTR DWORD, ; 实际读取的数量
    lpOverlapped:PTR DWORD        ; 异步信息的指针

```

其中 hFile 参数是 CreateFile 函数返回的已经打开的文件句柄；lpBuffer 参数指向用于接收读取的数据的缓冲区；nNumberOfBytesToRead 参数指定了最多要读取多少字节的数据；lpNumberOfBytesRead 参数是一个指针，指向一个整数变量，函数返回的时候会在此填入最终实际读取的字节数；lpOverlapped 参数是可选的，它指向一个描述如何在异步操作方式下读取文件的数据结构，在同步模式下（这是默认使用的方式）应设为 NULL（0）。如果函数失败则返回 0。

ReadFile 内部维护了一个指向当前文件位置的指针。如果针对同一个文件句柄多次调用 ReadFile 函数，则 ReadFile 能够记住上次读之后的位置并从该位置开始继续读。ReadFile 也可在异步模式下运行，这意味着程序可以不必一直等待直到操作完成。

WriteFile 函数

WriteFile 函数把数据写入文件，它使用一个输出句柄，这个句柄可以是一个控制台的屏幕缓冲区句柄，也可以是一个文本文件（或二进制文件）的句柄。数据的写入位置取决于文件内部的读写指针。写操作完成后，读写指针将根据实际写入的字节数做调整。函数原型如下所示：

```

WriteFile PROTO,
    hFile:HANDLE,                ; 输出句柄
    lpBuffer:PTR BYTE,           ; 缓冲区指针
    nNumberOfBytesToWrite:DWORD,  ; 缓冲区大小
    lpNumberOfBytesWritten:PTR DWORD, ; 实际写入的字节数
    lpOverlapped:PTR DWORD        ; 异步信息的指针

```

其中 hFile 是以前打开的文件的句柄；lpBuffer 是指向存放要写入文件的数据缓冲区的指针；nNumberOfBytesToWrite 指定了要写入文件多少个字节；lpNumberOfBytesWritten 参数是一个指针，指向一个整数变量，函数返回的时候会在此填入最终实际写入的字节数；lpOverlapped 是指向异步操作信息的指针，对于同步操作应设置为 NULL。如果函数失败则返回 0。

SetFilePointer 函数

SetFilePointer 函数移动一个已打开的文件的读写指针位置，这个函数可以用来在文件最后添加数据或者对文件进行随机记录处理操作：

```

SetFilePointer PROTO,
    hFile:HANDLE,                ; 文件句柄
    lDistanceToMove:SDWORD,       ; 文件指针要移动多少个字节
    lpDistanceToMoveHigh:PTR SDWORD, ; 指向包含移动字节数高位部分的指针
    dwMoveMethod:DWORD           ; 开始位置

```

如果函数失败则返回 0。dwMoveMethod 参数指定了从哪个位置开始移动读写指针，它可以有 3 种取值：FILE_BEGIN，FILE_CURRENT 和 FILE_END。要移动的字节数是一个 64 位的带符号整数，它被分为两个部分：

- lpDistanceToMove——低 32 位；
- lpDistanceToMoveHigh——指向一个变量，里面存放高 32 位。

如果 lpDistanceToMoveHigh 的值是 NULL（0）的话，则移动文件指针的时候函数只使用 lpDistanceToMove 中的值。下面的例子代码准备在文件尾部添加数据：

```

INVOKE SetFilePointer,
    fileHandle,          ; 文件句柄
    0,                  ; 要移动字节数的低 32 位
    0,                  ; 要移动字节数的高 32 位
    FILE_END            ; 移动模式

```

参见 AppendFile.asm 例子程序。

11.1.7 Irvine32 库的文件 I/O 过程

Irvine32 库包含了一些用于文件输入输出的过程，在第 5 章已经介绍过了。这些过程是对本章中已经介绍过的 Win32 API 函数的封装。下面列出了 CreateOutputFile, OpenFile, WriteToFile, ReadFromFile 和 CloseFile 的源码。

```

;-----
CreateOutputFile PROC
;
; Creates a new file and opens it in output mode.
; Receives: EDX points to the filename.
; Returns: If the file was created successfully, EAX
;          contains a valid file handle. Otherwise, EAX
;          equals INVALID_HANDLE_VALUE.
;-----
    INVOKE CreateFile,
        edx, GENERIC_WRITE, DO_NOT_SHARE, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
    ret
CreateOutputFile ENDP

;-----
OpenFile PROC
;
; Opens a new text file and opens for input.
; Receives: EDX points to the filename.
; Returns: If the file was opened successfully, EAX
;          contains a valid file handle. Otherwise, EAX equals
;          INVALID_HANDLE_VALUE.
;-----
    INVOKE CreateFile,
        edx, GENERIC_READ, DO_NOT_SHARE, NULL,
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
    ret
OpenFile ENDP

;-----
WriteToFile PROC
;
; Writes a buffer to an output file.
; Receives: EAX = file handle, EDX = buffer offset,
;          ECX = number of bytes to write
; Returns: EAX = number of bytes written to the file.
; If the value returned in EAX is less than the
; argument passed in ECX, an error likely occurred.
;-----
.data
WriteToFile_1 DWORD ?          ; number of bytes written
.code
    INVOKE WriteFile,
        eax,                  ; write buffer to file
        edx,                  ; file handle
        ; buffer pointer

```

```

        ecx,          ; number of bytes to write
        ADDR WriteToFile_1, ; number of bytes written
        0             ; overlapped execution flag
    mov     eax,WriteToFile_1 ; return value
    ret
WriteToFile ENDP
;-----
ReadFromFile PROC
;
; Reads an input file into a buffer.
; Receives: EAX = file handle, EDX = buffer offset,
; ECX = number of bytes to read
; Returns: If CF = 0, EAX = number of bytes read; if
; CF = 1, EAX contains the system error code returned
; by the GetLastError Win32 API function.
;-----
.data
ReadFromFile_1 DWORD ? ; number of bytes read
.code
    INVOKE ReadFile,
        eax,          ; file handle
        edx,          ; buffer pointer
        ecx,          ; max bytes to read
        ADDR ReadFromFile_1, ; number of bytes read
        0             ; overlapped execution flag
    mov     eax,ReadFromFile_1
    ret
ReadFromFile ENDP
;-----
CloseFile PROC
;
; Closes a file using its handle as an identifier.
; Receives: EAX = file handle
; Returns: EAX = nonzero if the file is successfully
; closed.
;-----
    INVOKE CloseHandle, eax
    ret
CloseFile ENDP

```

11.1.8 测试文件 I/O 过程

演示创建文件的例子程序

下面的程序创建了一个输出文件，要求用户输入一段文本，然后把文本写入输出文件并报告已写入的字节数，最后关闭文件。程序在尝试创建文件之后检查是否发生了错误：

```

TITLE Creating a File             (CreateFile.asm)
INCLUDE Irvine32.inc

BUFFER_SIZE = 501

.data
buffer BYTE BUFFER_SIZE DUP(?)
filename BYTE "output.txt",0
fileHandle HANDLE ?
stringLength DWORD ?
bytesWritten DWORD ?
str1 BYTE "Cannot create file",0dh,0ah,0
str2 BYTE "Bytes written to file [output.txt]:",0
str3 BYTE "Enter up to 500 characters and press"
    BYTE "[Enter]: ",0dh,0ah,0

```

```

.code
main PROC
; 创建一个新的文本文件
    mov     edx,OFFSET filename
    call    CreateOutputFile
    mov     fileHandle,eax

; 检查错误
    cmp     eax, INVALID_HANDLE_VALUE      ; 是否发生了错误?
    jne     file_ok                        ; 否: 跳过
    mov     edx,OFFSET str1                ; 显示错误
    call    WriteString
    jmp     quit
file_ok:
; 要求用户输入一个字符串
    mov     edx,OFFSET str3                ; "Enter up to ...."
    call    WriteString
    mov     ecx,BUFFER_SIZE               ; 输入一个字符串
    mov     edx,OFFSET buffer
    call    ReadString
    mov     stringLength,eax              ; 计算输入的字符的数目

; 把缓冲区写入输出文件
    mov     eax,fileHandle
    mov     edx,OFFSET buffer
    mov     ecx,stringLength
    call    WriteToFile
    mov     bytesWritten,eax              ; 保存返回值
    call    CloseFile

; 显示返回值
    mov     edx,OFFSET str2                ; "Bytes written"
    call    WriteString
    mov     eax,bytesWritten
    call    WriteDec
    call    CrLf

quit:
    exit
main ENDP
END main

```

演示读取文件的例子程序

下面的程序打开一个文件用于输入，把它的内容读入一个缓冲区，然后显示缓冲区的内容。调用的所有过程都是 Irvine32 库中的过程：

```

TITLE Reading a File                                (ReadFile.asm)
; Opens, reads, and displays a text file using
; procedures from Irvine32.lib.

INCLUDE Irvine32.inc
INCLUDE macros.inc
BUFFER_SIZE = 5000

.data
buffer BYTE BUFFER_SIZE DUP(?)
filename BYTE 80 DUP(0)
fileHandle HANDLE ?

.code
main PROC
; 允许用户输入一个文件名
    mWrite "Enter an input filename: "
    mov     edx,OFFSET filename
    mov     ecx,SIZEOF filename

```

```

    call ReadString
; 打开文件用于输入
    mov     edx,OFFSET filename
    call    OpenInputFile
    mov     fileHandle,eax
; 检查错误
    cmp     eax,INVALID_HANDLE_VALUE    ; 打开文件错误?
    jne     file_ok                     ; 否: 跳过
    mWrite  <"Cannot open file",0dh,0ah>
    jmp     quit                        ; 退出
file_ok:
; 把文件内容读入一个缓冲区
    mov     edx,OFFSET buffer
    mov     ecx,BUFFER_SIZE
    call    ReadFromFile
    jnc     check_buffer_size           ; 读取错误?
    mWrite  "Error reading file. "      ; 是: 显示一条错误信息
    call    WriteWindowsMsg
    jmp     close_file
check_buffer_size:
    cmp     eax,BUFFER_SIZE            ; 缓冲区足够大吗?
    jb     buf_size_ok                 ; 是
    mWrite  <"Error: Buffer too small for the file",0dh,0ah>
    jmp     quit                        ; 退出
buf_size_ok:
    mov     buffer[eax],0               ; 插入一个 null(0) 结束符
    mWrite  "File size: "
    call    WriteDec                    ; 显示文件的大小
    call    Crlf
; 显示缓冲区的内容
    mWrite  <"Buffer:",0dh,0ah,0dh,0ah>
    mov     edx,OFFSET buffer          ; 显示缓冲区的内容
    call    WriteString
    call    Crlf
close_file:
    mov     eax,fileHandle
    call    CloseFile
quit:
    exit
main ENDP
END main

```

如果文件无法打开, 程序就会报告一个类似下面的错误:

```

Enter an input filename: crazy.txt
Cannot open file

```

如果无法读取文件, 程序也会报告错误。假设如果程序使用了错误的文件句柄, 就会报告类似下面的错误:

```

Enter an input filename: infile.txt
Error reading file. Error 6: The handle is invalid.

```

也有可能文件太大而缓冲区太小, 这时也会报告一个错误:

```

Enter an input filename: infile.txt
Error: Buffer too small for the file

```

11.1.9 控制台窗口的操作

通过 Win32 API 可以对控制台窗口及其缓冲区进行非常多的控制操作，图 11.1 表明屏幕缓冲区有可能大于控制台窗口当前显示的行数。控制台窗口就像一个“视图”，仅仅显示部分缓冲区的内容。

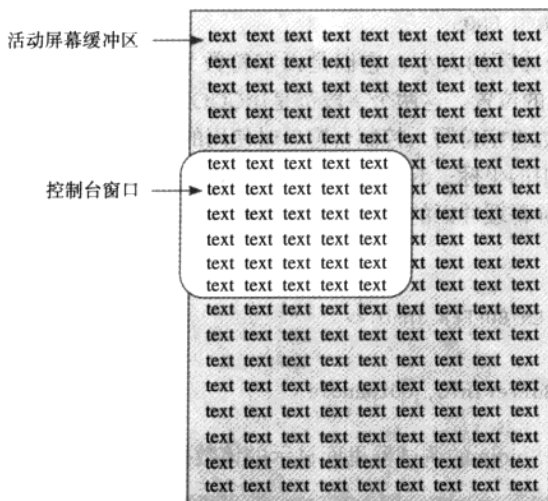


图 11.1 屏幕缓冲区和控制台窗口的关系

有几个函数可以影响控制台窗口以及它在与之关联的屏幕缓冲区中的位置：

- `SetConsoleWindowInfo` 函数设置控制台窗口在与之相关的屏幕缓冲区中的大小和位置。
- `GetConsoleScreenBufferInfo` 函数返回控制台窗口矩形在与之相关屏幕缓冲区中的坐标（当然这个函数还返回其他的一些数据）。
- `SetConsoleCursorPosition` 函数把光标定位到屏幕缓冲区中的指定位置，如果这个位置不在可视区域内，控制台窗口会滚动直到光标可见为止。
- `ScrollConsoleScreenBuffer` 函数将屏幕缓冲区中的部分或者全部文本移动出去，这个函数可能会影响到控制台窗口中显示的文本。

SetConsoleTitle 函数

`SetConsoleTitle` 函数允许改变控制台窗口的标题栏文字，例如：

```
.data
titleStr BYTE "Console title",0
.code
INVOKE SetConsoleTitle, ADDR titleStr
```

GetConsoleScreenBufferInfo 函数

`GetConsoleScreenBufferInfo` 函数返回控制台窗口当前状态的相关信息，这个函数有两个参数：控制台屏幕的句柄以及指向一个结构的指针，这个结构将由下列函数填写：

```
GetConsoleScreenBufferInfo PROTO,
    hConsoleOutput:HANDLE,
    lpConsoleScreenBufferInfo:PTR CONSOLE_SCREEN_BUFFER_INFO
```


下面是 CONSOLE_SCREEN_BUFFER_INFO 结构的定义：

```
CONSOLE_SCREEN_BUFFER_INFO STRUCT
    dwSize          COORD <>
    dwCursorPosition COORD <>
    wAttributes      WORD ?
    srWindow         SMALL_RECT <>
    dwMaximumWindowSize COORD <>
CONSOLE_SCREEN_BUFFER_INFO ENDS
```

dwSize 字段中返回屏幕缓冲区的大小，它们是以字符数为单位进行度量的行数和列数；dwCursorPosition 字段返回光标的位置。这两个字段都是用COORD结构定义的。wAttributes 返回 WriteConsole 和 WriteFile 等函数将字符写入控制台窗口时使用的前景和背景颜色；srWindow 字段返回控制台窗口在屏幕缓冲区中的坐标；dwMaximumWindowSize 字段返回控制台窗口的最大可能尺寸，这个数据是根据当前屏幕缓冲区的大小、字体和视频显示的尺寸综合得出的。下面是使用该函数的简单例子：

```
.data
consoleInfo CONSOLE_SCREEN_BUFFER_INFO <>
outHandle HANDLE ?
.code
INVOKE GetConsoleScreenBufferInfo, outHandle,
      ADDR consoleInfo
```

图 11.2 是 Microsoft Visual Studio 的调试器里面显示的该数据结构的图例。

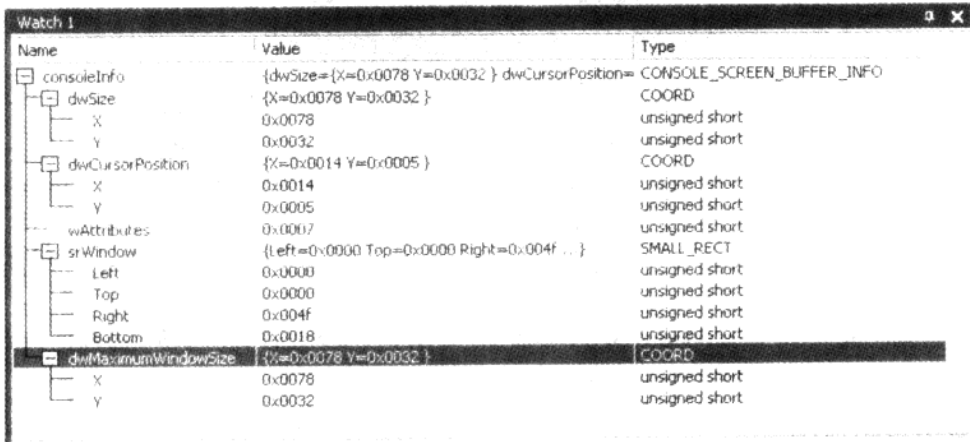


图 11.2 CONSOLE_SCREEN_BUFFER_INFO 结构

SetConsoleWindowInfo 函数

SetConsoleWindowInfo 函数用来设置控制台窗口在与其相关的屏幕缓冲区中的大小和位置。

函数原型如下所示：

```
SetConsoleWindowInfo PROTO,
hConsoleOutput:HANDLE,          ; 屏幕输出句柄
bAbsolute:DWORD,                ; 坐标类型
lpConsoleWindow:PTR SMALL_RECT ; 指向窗口矩形的坐标
```

bAbsolute 参数指定了 lpConsoleWindow 参数指定的坐标应如何解释，如果 bAbsolute 的值是 TRUE，那么坐标代表控制台窗口新的左上角和右下角的位置；如果 bAbsolute 的值是 FALSE，那么新的坐标将被加到当前控制台坐标上面使用。

下面的 Scroll.asm 程序在屏幕缓冲区上显示 50 行文本，然后改变控制台窗口的位置和大小，这达到了有效地回滚文字的效果。程序中用到了 SetConsoleWindowInfo 函数：

```
TITLE Scrolling the Console Window                (Scroll.asm)
INCLUDE Irvine32.inc

.data
message BYTE " : This line of text was written "
        BYTE "to the screen buffer",0dh,0ah
messageSize DWORD ($-message)

outHandle    HANDLE 0                        ; 标准输出句柄
bytesWritten  DWORD ?                        ; 已写的字节数
lineNum      DWORD 0
windowRect   SMALL_RECT <0,0,60,11>        ; 左、上、右、下

.code
main PROC
    INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    mov outHandle,eax

    .REPEAT
        mov     eax,lineNum
        call    WriteDec                ; 显示每行的行号
        INVOKE WriteConsole,
            outHandle,                    ; 控制台输出句柄
            ADDR message,                ; 字符串指针
            messageSize,                  ; 字符串长度
            ADDR bytesWritten,            ; 返回已写的字节数
            0,                             ; 未用
            inc lineNum                    ; 下一行的行号
    .UNTIL lineNum > 50

    ; Resize and reposition the console window relative to the
    ; screen buffer.
    INVOKE SetConsoleWindowInfo,
        outHandle,
        TRUE,
        ADDR windowRect; window rectangle

    call    Readchar                    ; 等待按键
    call    Clrscr                       ; 清除屏幕缓冲区
    call    Readchar                    ; 等待第二次按键

    INVOKE ExitProcess,0
main ENDP
END main
```

由于集成开发(编辑)环境可能会影响控制台窗口的外观和动作，因此最好是在 MS-Windows 的资源管理器中或命令行上直接运行这个程序，而不是通过 IDE 运行。另外，在程序运行后读者必须按下两次键盘：第一次是为了清除屏幕缓冲区，另一次程序将退出（这个功能是为了测试而加上的）。

SetConsoleScreenBufferSize 函数

SetConsoleScreenBufferSize 函数允许将屏幕缓冲区的大小设置为 X 列 Y 行。函数原型如下所示：

```
SetConsoleScreenBufferSize PROTO,
    hConsoleOutput:HANDLE,    ; 到屏幕缓冲区的句柄
    dwSize:COORD              ; 新的屏幕缓冲区的大小
```

11.1.10 光标的控制

Win32 API 提供了设置光标的大小、可见性和屏幕位置的函数, 与这些函数相关的一个重要的数据结构称为 `CONSOLE_CURSOR_INFO`, 其中包含了光标大小和可见性等相关信息:

```
CONSOLE_CURSOR_INFO STRUCT
    dwSize    DWORD ?
    bVisible  DWORD ?
CONSOLE_CURSOR_INFO ENDS
```

`dwSize` 字段是光标占字符方格大小的百分比 (1~100); `bVisible` 等于 `TRUE` (1) 的时候, 光标是可见的。

GetConsoleCursorInfo 函数

`GetConsoleCursorInfo` 函数返回控制台光标的大小和可见性等信息, 调用的时候需要传递给函数一个指向 `CONSOLE_CURSOR_INFO` 结构的指针:

```
GetConsoleCursorInfo PROTO,
    hConsoleOutput:HANDLE,
    lpConsoleCursorInfo:PTR CONSOLE_CURSOR_INFO
```

在默认状态下, 光标尺寸等于 25, 也就是说光标占一个字符方格的 25% 大小。

SetConsoleCursorInfo 函数

`SetConsoleCursorInfo` 函数设置控制台光标的大小和可见性。同样, 调用的时候需要传递给函数一个 `CONSOLE_CURSOR_INFO` 结构的指针:

```
SetConsoleCursorInfo PROTO,
    hConsoleOutput:HANDLE,
    lpConsoleCursorInfo:PTR CONSOLE_CURSOR_INFO
```

SetConsoleCursorPosition 函数

`SetConsoleCursorPosition` 函数设置光标的 X 和 Y 坐标位置, 调用的时候需要传递给函数一个 `COORD` 结构和控制台输出句柄:

```
SetConsoleCursorPosition PROTO,
    hConsoleOutput:DWORD,          ; 输出句柄
    dwCursorPosition:COORD        ; 屏幕 X 和 Y 坐标位置
```

11.1.11 文本颜色的控制

有两种方式用来设置控制台窗口中的文本颜色, 可以调用 `SetConsoleTextAttribute` 函数来设置当前的文本颜色, 这样以后输出的所有文本的颜色都会改变。或者, 也可以使用 `WriteConsoleOutputAttribute` 函数来设置指定位置字符位置的颜色属性。`GetConsoleScreenBufferInfo` 函数 (11.1.9 节) 返回当前屏幕的颜色以及其他的控制台信息。

SetConsoleTextAttribute 函数

`SetConsoleTextAttribute` 函数用来设置以后输出到控制台窗口的字符的前景和背景颜色。函数原型如下所示:

```
SetConsoleTextAttribute PROTO,
    hConsoleOutput:HANDLE,          ; 控制输出句柄
    wAttributes:WORD                ; 颜色属性
```

颜色值存放于 `wAttributes` 参数的低位,其定义和 15.3.2 节中演示视频 BIOS 使用时的取值含义相同。

WriteConsoleOutputAttribute 函数

`WriteConsoleOutputAttribute` 函数复制一个颜色属性数组的值到控制台屏幕缓冲区中连续位置的字符格中,字符格的位置是可以指定的。函数原型如下所示:

```
WriteConsoleOutputAttribute PROTO,
    hConsoleOutput:DWORD,          ; 输出句柄
    lpAttribute:PTR WORD,          ; 要输出的属性
    nLength:DWORD,                 ; 颜色属性的数量
    dwWriteCoord:COORD,            ; 首个字符格的坐标
    lpNumberOfAttrsWritten:PTR DWORD ; 实际输出的数量
```

`lpAttribute` 指向一个颜色属性数组,每个数组元素的低字节中存放颜色值;`nLength` 是数组的长度;`dwWriteCoord` 是屏幕中要接收这些属性的字符格的起始坐标;`lpNumberOfAttrsWritten` 指向一个变量,函数返回的时候将在此填写实际被设置了颜色属性的字符格的数量。

WriteColors 例子程序

为了演示如何使用颜色属性,例子程序 `WriteColors.asm` 创建了一个字符数组和一个属性数组,属性数组里面的每个属性对应字符数组里面的一个字符。程序调用 `WriteConsoleOutputAttribute` 函数把颜色属性复制到屏幕缓冲区中,然后调用 `WriteConsoleOutputCharacter` 函数把字符复制到屏幕缓冲区的同样位置:

```
TITLE Writing Text Colors                (WriteColors.asm)

INCLUDE Irvine32.inc

.data
outHandle    HANDLE ?
cellsWritten DWORD ?
xyPos        COORD <10,2>

; 字符代码数组
buffer BYTE 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
        BYTE 16,17,18,19,20
BufSize    DWORD ($-buffer)
; Array of attributes:
attributes WORD 0Fh,0Eh,0Dh,0Ch,0Bh,0Ah,9,8,7,6
        WORD 5,4,3,2,1,0F0h,0E0h,0D0h,0C0h,0B0h

.code
main PROC
; 获取控制台标准输出的句柄
    INVOKE GetStdHandle,STD_OUTPUT_HANDLE
    mov outHandle,eax

; 设置相邻连续字符格的颜色
    INVOKE WriteConsoleOutputAttribute,
        outHandle, ADDR attributes,
        BufSize, xyPos,
        ADDR cellsWritten

; 输出 1 到 20 的字符代码
    INVOKE WriteConsoleOutputCharacter,
        outHandle, ADDR buffer, BufSize,
        xyPos, ADDR cellsWritten

    INVOKE ExitProcess,0                ; 结束程序
main ENDP
END main
```

图 11.3 是程序输出结果的屏幕截图，在图中可以看出字符码 1~20 被显示为图形字符，每个字符的显示颜色是各不相同的（虽然在纸张上面没有印出色彩来）。

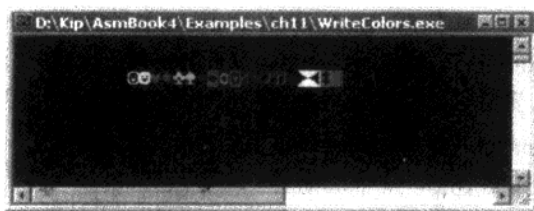


图 11.3 WriteColors 程序的输出

11.1.12 时间和日期函数

Win32 API 提供了大量的时间和日期函数。对于初学者来说，可以用它们来获取或者设置当前日期和时间。本节的内容仅仅示范了一小部分的时间和日期函数，读者可以查阅 Platform SDK 文档，进一步了解表 11.9 中列出的这些函数。

表 11.9 Win32 日期和时间函数^①

函 数	说 明
CompareFileTime	比较两个 64 位的文件时间
DosDateTimeToFileTime	把 MS-DOS 日期和时间转换成 64 位的文件时间
FileTimeToDosDateTime	把 64 位的文件时间转换成 MS-DOS 时间
FileTimeToLocalFileTime	把 UTC 文件时间转换成本地文件时间
FileTimeToSystemTime	把 64 位文件时间转换成系统时间格式
GetFileTime	获取文件创建、最后一次访问和最后一次修改的时间
GetLocalTime	获取本地的系统日期和时间
GetSystemTime	获取 UTC 格式的系统日期和时间
GetSystemTimeAdjustment	确定系统是否定期对系统时钟进行调整至与外部时钟信号源同步
GetSystemTimeAsFileTime	获取当前系统的 UTC 时间，以文件时间的格式
GetTickCount	获取系统启动以来的毫秒数
GetTimeZoneInformation	获取当前时区信息
LocalFileTimeToFileTime	把本地文件时间转换成 UTC 文件时间
SetFileTime	设置文件创建、最后一次访问或最后一次修改的时间
SetLocalTime	设置本地系统时间和日期
SetSystemTime	设置系统 UTC 时间和日期
SetSystemTimeAdjustment	允许或禁止系统日时钟与一个外部的时钟信号源同步
SetTimeZoneInformation	设置时区信息
SystemTimeToFileTime	把系统时间格式转换成文件时间格式
SystemTimeToTzSpecificLocalTime	将系统 UTC 时间转换成特定时区中的当地时间

SYSTEMTIME 结构：与日期和时间相关的 Win32 API 函数常常会用到 SYSTEMTIME 结构：

^① 来源：Microsoft MSDN Windows SDK 文档。

```

SYSTEMTIME STRUCT
    wYear WORD ?           ; 年 (4 位数)
    wMonth WORD ?          ; 月 (1~12)
    wDayOfWeek WORD ?      ; 星期 (0~6)
    wDay WORD ?            ; 日 (1~31)
    wHour WORD ?           ; 小时 (0~23)
    wMinute WORD ?         ; 分钟 (0~59)
    wSecond WORD ?         ; 秒 (0~59)
    wMilliseconds WORD ?   ; 毫秒 (0~999)
SYSTEMTIME ENDS

```

wDayOfWeek 字段的值含义为: 0 = 星期日, 1 = 星期一, 依次类推。由于计算机的内部时钟是每经过一个时间间隔定期与外部时钟源同步更新时钟的, 所以 wMilliseconds 字段的值并不是完全精确的。

GetLocalTime 和 SetLocalTime 函数

GetLocalTime 函数获取当前的日期和时间, 该时间已经按照系统日期和时间转换成了本地地区的时间。当调用这个函数的时候, 需要传递给函数一个指向 SYSTEMTIME 结构的指针:

```

GetLocalTime PROTO,
    pSystemTime:PTR SYSTEMTIME

```

下面是 GetLocalTime 函数的调用示例:

```

.data
sysTime SYSTEMTIME <>
.code
INVOKE GetLocalTime, ADDR sysTime

```

SetLocalTime 函数设置系统的当前时间和日期, 调用的时候也需要传递一个指向 SYSTEMTIME 结构的指针, 其中包含了要设置的时间值:

```

SetLocalTime PROTO,
    pSystemTime:PTR SYSTEMTIME

```

如果函数执行成功, 返回值是非 0 值; 如果执行失败, 函数返回 0。

GetTickCount 函数

GetTickCount 函数返回系统启动以来所经过的毫秒数:

```

GetTickCount PROTO           ; 返回值在 EAX 中

```

由于计数值是一个双字, 所以当系统连续运行 49.7 天后, 计数值将归 0。可以在一个循环中使用这个函数来监控经过的时间, 并在某个时间到达的时候退出循环。

下面的例子程序 Timer.asm 测量两次调用 GetTickCount 之间经过的时间, 并检查时间计数器值是否发生了回滚 (超过了 49.7 天)。类似的代码可以用在很多不同的程序中:

```

TITLE Calculate Elapsed Time          (Timer.asm)
; Demonstrate a simple stopwatch timer, using
; the Win32 GetTickCount function.

INCLUDE Irvine32.inc
INCLUDE macros.inc

.data
startTime DWORD ?
.code

```

```

main PROC
    INVOKE GetTickCount          ; 获取起始的时间计数值
    mov     startTime,eax        ; 保存之
    ; Create a useless calculation loop.
    mov     ecx,10000100h
L1:  imul    ebx
    imul    ebx
    imul    ebx
    loop    L1

    INVOKE GetTickCount          ; 获取新的时间计数值
    cmp     eax,startTime        ; 小于起始时间?
    jb      error                ; 时间回滚了

    sub     eax,startTime        ; 获取经过的毫秒数
    call    WriteDec             ; 显示
    mWrite <" milliseconds have elapsed",0dh,0ah>
    jmp     quit

error:
    mWrite "Error: GetTickCount invalid--system has"
    mWrite <"been active for more than 49.7 days",0dh,0ah>
quit:
    exit
main ENDP
END main

```

Sleep 函数

程序有时需要暂停或延迟一小段时间。可以编写一个循环达到延时的目的，但循环的执行时间在不同的处理器上是不同的。除此之外，循环计算还会大量消耗 CPU 资源，同时降低其他程序的执行速度。Sleep 函数挂起当前的线程指定的毫秒数：

```

Sleep Proto,
    dwMilliseconds:DWORD

```

(由于大多数汇编语言程序通常是单线程的，因此这里假设一个线程就代表一个程序。)线程在睡眠的时候不会占用处理器时间。

GetDateTime 子程序

Irvine32 库中的 GetDateTime 过程返回了一个 64 位的整数，这个数值是自 1601 年 1 月 1 日开始的以 100 ns 为单位的计数值。这看起来有些奇怪，因为在那个时候人们还不知道计算机是什么，但是 Microsoft 却使用这个数值来跟踪文件的日期和时间。Win32 SDK 文档中建议读者使用下面的步骤来准备系统日期和时间，以便进行其他的日期和时间计算：

1. 调用一个函数（例如 GetLocalTime）来填写 SYSTEMTIME 结构。
2. 用 SystemTimeToFileTime 函数把 SYSTEMTIME 结构转换到 FILETIME 结构。
3. 把 FILETIME 结构中的结果复制到一个 64 位的 QWORD 中。

FILETIME 结构把一个 64 位的 QWORD 值划分为两个 DWORD 值：

```

FILETIME STRUCT
    loDateTime DWORD ?
    hiDateTime DWORD ?
FILETIME ENDS

```

下面的 GetDateTime 接收一个指向 64 位 QWORD 变量的参数，并把当前的日期和时间存储在这个变量中，时间格式使用 FILETIME 格式：

```

;-----
GetDateTime PROC,
    pStartTime:PTR QWORD
    LOCAL sysTime:SYSTEMTIME, flTime:FILETIME
;
; Gets and saves the current local date/time as a
; 64-bit integer (in the Win32 FILETIME format).
;-----
; 获取本地系统时间
    INVOKE GetLocalTime,
        ADDR sysTime

; 把 SYSTEMTIME 转换成 FILETIME
    INVOKE SystemTimeToFileTime,
        ADDR sysTime,
        ADDR flTime

; 把 FILETIME 转换成一个 64 位的整数
    mov     esi,pStartTime
    mov     eax,flTime.loDateTime
    mov     DWORD PTR [esi],eax
    mov     eax,flTime.hiDateTime
    mov     DWORD PTR [esi+4],eax
    ret
GetDateTime ENDP

```

由于返回的时间值是一个 64 位的整数，因此可以使用 7.5 节讲述的扩展精度算术运算技术进行日期的算术运算。

11.1.13 本节习题

1. 哪条 LINK 命令可以把程序的目标文件类型指定为 Win32 控制台程序？
2. （对/错）以 W 字母结尾的函数（如 WriteConsoleW）是设计用来处理宽字符集（Unicode 等 16 位字符）的。
3. （对/错）Unicode 字符集是 Windows 98 操作系统内置的字符集。
4. （对/错）ReadConsole 函数从输入缓冲区读取鼠标的信息。
5. （对/错）Win32 控制台输入函数可以检测到用户何时改变了控制台窗口的大小。
6. 说出下列标准的 Windows 类型分别对应于哪些 MASM 数据类型？

```

BOOL
COLORREF
HANDLE
LPSTR
WPARAM

```

7. 哪个 Win32 函数返回标准输入的句柄？
8. 哪个高级 Win32 函数从键盘读入一个字符串并把字符串放到一个缓冲区中？
9. 试举一个使用 ReadConsole 函数的例子。
10. 描述一下 COORD 结构。
11. 试举一个使用 WriteConsole 函数的例子。
12. 试举一个使用 CreateFile 函数来打开一个现存文件以便进行读数据的例子。
13. 试举一个例子，使用 CreateFile 函数建立一个普通属性的新文件，如果文件存在，则将其覆盖。
14. 试举一个使用 ReadFile 函数的例子。
15. 试举一个使用 WriteFile 函数的例子。

- 16. 哪个 Win32 函数用来移动文件读写指针？
- 17. 哪个 Win32 函数用来改变控制台窗口标题栏上面的文字？
- 18. 哪个 Win32 函数用来改变屏幕缓冲区的尺寸？
- 19. 哪个 Win32 函数用来改变光标的大小？
- 20. 哪个 Win32 函数用来设置以后要输出的文本的颜色？
- 21. 哪个 Win32 函数用来把一个颜色属性数组复制到控制台屏幕缓冲区中连续的字符格上？
- 22. 哪个 Win32 函数用来让程序暂停指定毫秒数的时间？

11.2 编写 Windows 图形界面应用程序

在本节中，我们来演示如何写出一个简单的 Windows 下的图形界面应用程序，这个程序将创建并显示一个主窗口，显示一些消息框，而且可以响应鼠标动作。下面的内容仅仅是一个简要的介绍，因为即使是要描述清楚一个最简单的 Windows 应用程序中的事件，也需要至少一整章的篇幅。如果读者需要更详尽的信息，请参阅 Platform SDK 的文档。另一份很好的文献资料是 Charles Petzold 所著的 *Programming in Windows:The Definitive Guide to the Win32 API* 一书。

表 11.10 列出了构建该程序时使用的各种库和包含文件，可以使用本书附带代码 Examples \Ch11\WinApp 目录下的 Visual Studio 项目文件构建和运行该程序。

表 11.10 构建 WinApp 程序时需要的文件

文 件 名	说 明
WinApp.asm	程序源代码
GraphWin.inc	包括程序使用的数据结构、常量定义、函数原型声明等内容的包含文件
kernel32.lib	Windows API 库文件，前面的章节中也使用了这个文件
user32.lib	另一个 Windows API 库文件

编译时应使用选项 /SUBSYSTEM:WINDOWS 代替我们在前面章节中使用的 /SUBSYSTEM:CONSOLE 选项。程序调用了两个标准 Windows 库文件：kernel32.lib 和 user32.lib 文件。

主窗口：程序将显示一个填满整个屏幕的主窗口。下面的图例已经缩小了，以便于在书中印刷。

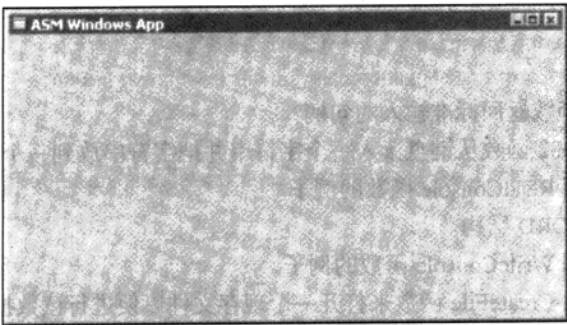


图 11.4 WinApp 程序的主启动程序

11.2.1 必须了解的数据结构

POINT 结构定义了以像素为单位的屏幕上某个点的 X 和 Y 坐标，它可以用来定位屏幕上的

某个对象的坐标，如图形对象、窗口、鼠标单击时的位置等：

```
POINT STRUCT
    ptX  DWORD ?
    ptY  DWORD ?
POINT ENDS
```

RECT 结构定义了一个矩形的边界，left 字段为矩形左边界的 X 坐标，top 字段为矩形顶边的 Y 坐标。类似地，right 和 bottom 字段的值定义了矩形右下角的坐标：

```
RECT STRUCT
    left      DWORD ?
    top       DWORD ?
    right     DWORD ?
    bottom    DWORD ?
RECT ENDS
```

MSGStruct 结构定义了 Windows 消息需要的相关数据：

```
MSGStruct STRUCT
    msgWnd      DWORD ?
    msgMessage  DWORD ?
    msgWparam   DWORD ?
    msgLparam   DWORD ?
    msgTime     DWORD ?
    msgPt       POINT <>
MSGStruct ENDS
```

WNDCLASS 结构定义了一个窗口类，程序中的每个窗口必须属于一个窗口类，所以每个程序必须为它的主窗口创建窗口类。在能够显示主窗口之前，窗口类必须首先在系统里面注册：

```
WNDCLASS STRUC
    style          DWORD ?           ; 窗口风格
    lpfnWndProc    DWORD ?           ; 窗口过程地址
    cbClsExtra     DWORD ?           ; 共享内存
    cbWndExtra     DWORD ?           ; 额外定义的数据
    hInstance      DWORD ?           ; 当前程序的句柄
    hIcon          DWORD ?           ; 图标句柄
    hCursor        DWORD ?           ; 光标句柄
    hbrBackground  DWORD ?           ; 背景画刷句柄
    lpszMenuName   DWORD ?           ; 菜单名称的指针
    lpszClassName  DWORD ?           ; 类名称的指针
WNDCLASS ENDS
```

以下是这些字段的简要介绍：

- style 是一些不同风格选项的组合，如 WS_CAPTION 和 WS_BORDER，这个字段影响窗口的外观和行为。
- lpfnWndProc 是指向一个子程序的指针，这个子程序在我们自己的程序中，用来接收由用户触发的事件消息。
- cbClsExtra 定义了这个窗口类所属的所有窗口都可以共享的内存，这个参数可以指定为 NULL。
- cbWndExtra 参数为每个窗口实例分配一些额外的内存。
- hInstance 参数用来保存当前运行程序的句柄。
- hIcon 和 hCursor 参数为当前程序使用的图标和光标句柄。
- hbrBackground 参数为背景颜色画刷的句柄。

- lpzMenuName 指向一个菜单名称字符串。
- lpzClassName 指向一个以 0 结尾的窗口类名称字符串。

11.2.2 MessageBox 函数

程序显示文本的最简单方法是把文本放到一个消息框中,消息框会弹出并直到用户按下了上面的某个按钮为止。Win32 API 中的 MessageBox 函数显示一个简单的消息框,它的函数原型如下所示:

```
MessageBox PROTO,  
    hWnd:DWORD,  
    lpText:PTR BYTE,  
    lpCaption:PTR BYTE,  
    uType:DWORD
```

hWnd 是当前窗口的句柄;lpText 指向要在消息框中显示的以 0 结尾的字符串;lpCaption 指向要显示在消息框标题栏上的以 0 结尾的字符串;style^①参数是一个整数,用来描述消息框中的图标(可选)和按钮(必选)的样式。按钮由 MB_OK 或 MB_YESNO 等常量定义,图标由 MB_ICONQUESTION 等常量定义。当想要显示一个消息框的时候,可以把这些常量加在一起以便同时显示图标和按钮:

```
INVOKE MessageBox, hWnd, ADDR QuestionText,  
    ADDR QuestionTitle, MB_OK + MB_ICONQUESTION
```

11.2.3 WinMain 过程

每个 Windows 应用程序都需要一个启动过程,通常名为 WinMain,它负责以下的工作:

- 获取当前程序的句柄。
- 装载程序使用的图标和鼠标光标。
- 注册主窗口使用的窗口类,并且指定用来接收窗口事件消息的过程。
- 创建主窗口。
- 显示并更新主窗口。
- 开始一个消息循环来接收和分派处理消息,循环会一直持续到用户关闭了应用程序窗口。

WinMain 中包含一个消息循环,使用 GetMessage 从程序的消息队列中返回下一个可用的消息。如果 GetMessage 取到 WM_QUIT 消息,那么函数返回 0,通知 WinMain 终止程序。对于其他消息,WinMain 会把它们传递给 DispatchMessage 函数,由该函数把消息分发给程序的 WinProc 过程。要了解更多关于消息方面的知识,可在 Platform SDK 文档中搜索“Windows Messages”。

11.2.4 WinProc 过程

WinProc 过程接收并处理所有和窗口相关的事件消息,大部分的事件是由用户单击、拖动鼠标或按下了一个键盘按键等动作而引起的。该过程的任务是解译每个消息。如果某个消息是可以辨认的,那么运行和该消息对应的任务。下面是过程的声明:

```
WinProc PROC,  
    hWnd:DWORD,                ; 窗口句柄  
    localMsg:DWORD,            ; 消息 ID  
    wParam:DWORD,              ; 参数 1 (可变)  
    lParam:DWORD               ; 参数 2 (可变)
```

① 此处原文如此,疑为 uType——编者注。

根据不同的消息 ID，第 3 个参数和第 4 个参数的含义是不同的。如当鼠标被按下的时候，lParam 参数里面包含的是鼠标按下点的 X 和 Y 坐标。在下面马上要看到的例子中，WinProc 过程处理了三种消息：

- WM_LBUTTONDOWN，用户按下鼠标左键的时候产生。
- WM_CREATE，表明主窗口刚刚被创建。
- WM_CLOSE，表明主窗口即将被关闭。

举例来说，过程中的下面几行代码处理 WM_LBUTTONDOWN 消息，处理方法是调用 MessageBox 函数向用户显示一条提示信息：

```
.IF eax == WM_LBUTTONDOWN
    INVOKE MessageBox, hWnd, ADDR PopupText,
        ADDR PopupTitle, MB_OK
    jmp WinProcExit
```

用户看到的结果如图 11.5 所示。任何我们不想处理的消息都应该传递给 Windows 的默认消息处理函数 DefWindowProc。

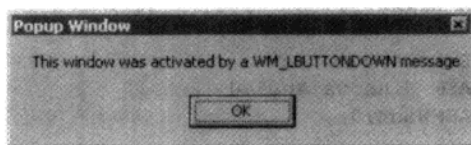


图 11.5 WinApp 程序的弹出消息框

11.2.5 ErrorHandler 过程

ErrorHandler 过程是可选的，如果程序的主窗口在注册和创建的过程中发生错误，就会调用这个过程。举例来说，调用 RegisterClass 函数时，如果窗口类成功注册，那么函数会返回一个非 0 值。若函数返回 0，则可调用 ErrorHandler 来显示出错信息并退出程序的执行：

```
INVOKE RegisterClass, ADDR MainWin
.IF eax == 0
    call ErrorHandler
    jmp Exit_Program
.ENDIF
```

ErrorHandler 过程完成下面几件重要的事情：

- 调用 GetLastError 函数获取系统错误代码。
- 调用 FormatMessage 函数获取操作系统格式化的出错信息字符串。
- 调用 MessageBox 显示包含出错信息的信息框。
- 调用 LocalFree 函数释放为出错信息字符串分配的内存。

11.2.6 程序清单

看到下面程序的篇幅请不要伤心！因为里面的大部分代码都可以在任何的 Windows 应用程序中重复使用：

```
TITLE Windows Application                                (WinApp.asm)

; This program displays a resizable application window and
```

```

; several popup message boxes. Special thanks to Tom Joyce
; for the first version of this program.

.386
.model flat,STDCALL
INCLUDE GraphWin.inc

;===== DATA =====
.data

AppLoadMsgTitle BYTE "Application Loaded",0
AppLoadMsgText  BYTE "This window displays when the WM_CREATE "
                  BYTE "message is received",0

PopupTitle BYTE "Popup Window",0
PopupText  BYTE "This window was activated by a "
              BYTE "WM_LBUTTONDOWN message",0
GreetTitle BYTE "Main Window Active",0
GreetText  BYTE "This window is shown immediately after "
              BYTE "CreateWindow and UpdateWindow are called.",0

CloseMsg  BYTE "WM_CLOSE message received",0

ErrorTitle  BYTE "Error",0
WindowName  BYTE "ASM Windows App",0
className   BYTE "ASMWin",0

; Define the Application's Window class structure.
MainWin WNDCLASS <NULL,WinProc,NULL,NULL,NULL,NULL, \
    COLOR_WINDOW,NULL,className>

msg      MSGStruct <>
winRect  RECT <>
hMainWnd DWORD ?
hInstance DWORD ?

;===== CODE =====
.code
WinMain PROC
; 获取当前进程的句柄
    INVOKE GetModuleHandle, NULL
    mov hInstance, eax
    mov MainWin.hInstance, eax
; 加载程序的光标和图标
    INVOKE LoadIcon, NULL, IDI_APPLICATION
    mov MainWin.hIcon, eax
    INVOKE LoadCursor, NULL, IDC_ARROW
    mov MainWin.hCursor, eax

; 注册窗口类
    INVOKE RegisterClass, ADDR MainWin
    .IF eax == 0
        call ErrorHandler
        jmp Exit_Program
    .ENDIF

; 创建应用程序的主窗口
    INVOKE CreateWindowEx, 0, ADDR className,
        ADDR WindowName,MAIN_WINDOW_STYLE,
        CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,

```

```

        CW_USEDEFAULT, NULL, NULL, hInstance, NULL
; 如果 CreateWindowEx 失败, 显示一条消息并退出
        .IF eax == 0
            call ErrorHandler
            jmp Exit_Program
        .ENDIF

; 保存窗口句柄, 显示并绘制窗口
        mov hMainWnd, eax
        INVOKE ShowWindow, hMainWnd, SW_SHOW
        INVOKE UpdateWindow, hMainWnd

; 显示欢迎消息
        INVOKE MessageBox, hMainWnd, ADDR GreetText,
            ADDR GreetTitle, MB_OK

; 开始程序的持续消息处理循环
Message_Loop:
    ; 从队列中获取下一条消息
    INVOKE GetMessage, ADDR msg, NULL, NULL, NULL

    ; 若无消息则退出
    .IF eax == 0
        jmp Exit_Program
    .ENDIF

    ; 把消息转发给程序的 WinProc 过程
    INVOKE DispatchMessage, ADDR msg
    jmp Message_Loop

Exit_Program:
    INVOKE ExitProcess, 0
WinMain ENDP

```

在上面的循环循环代码中, msg 结构被传递给 GetMessage 函数, 该函数将填写这个结构, 接下来该结构被传递给了 DispatchMessage 函数。

```

;-----
WinProc PROC,
    hWnd:DWORD, localMsg:DWORD, wParam:DWORD, lParam:DWORD
;
; The application's message handler, which handles
; application-specific messages. All other messages
; are forwarded to the default Windows message
; handler.
;-----
        mov eax, localMsg

        .IF eax == WM_LBUTTONDOWN        ; 鼠标按键消息?
            INVOKE MessageBox, hWnd, ADDR PopupText,
                ADDR PopupTitle, MB_OK
            jmp WinProcExit
        .ELSEIF eax == WM_CREATE          ; 创建窗口消息?
            INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,
                ADDR AppLoadMsgTitle, MB_OK
            jmp WinProcExit

```

```

.ELSEIF eax == WM_CLOSE      ; 关闭窗口消息?
    INVOKE MessageBox, hWnd, ADDR CloseMsg,
        ADDR WindowName, MB_OK
    INVOKE PostQuitMessage, 0
    jmp WinProcExit
.ELSE                        ; 其他消息?
    INVOKE DefWindowProc, hWnd, localMsg, wParam, lParam
    jmp WinProcExit
.ENDIF

WinProcExit:
    ret
WinProc ENDP

;-----
ErrorHandler PROC
; Display the appropriate system error message.
;-----
.data
pErrorMsg  DWORD ?      ; 指向错误消息的指针
messageID  DWORD ?
.code
    INVOKE GetLastError    ; 在 EAX 中返回消息 ID
    mov messageID, eax

; 获取对应的消息字符串
    INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
        FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageID, NULL,
        ADDR pErrorMsg, NULL, NULL

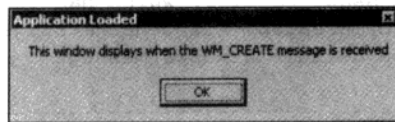
; 显示错误消息
    INVOKE MessageBox, NULL, pErrorMsg, ADDR ErrorTitle,
        MB_ICONERROR+MB_OK

; 释放消息字符串
    INVOKE LocalFree, pErrorMsg
    ret
ErrorHandler ENDP
END WinMain

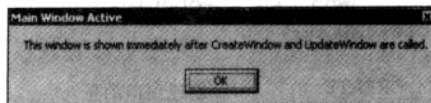
```

运行例子程序

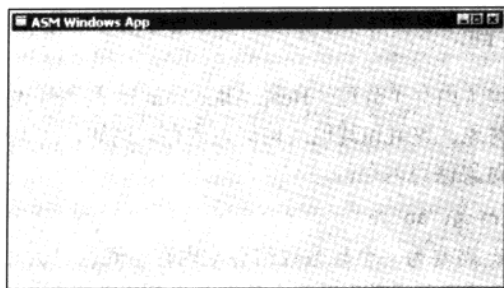
程序运行的时候首先显示下面的消息框:



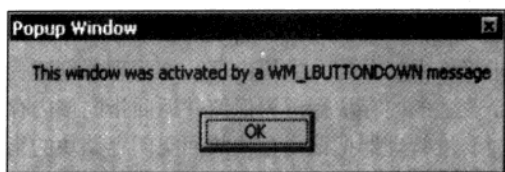
当用户按下了 OK 按钮来关闭这个 Application Loaded (程序已加载) 消息框后, 另一个消息框将会弹出:



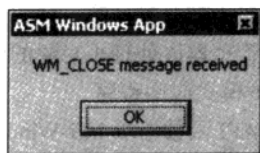
这是 Main Window Active (主窗口激活) 消息框, 按 OK 按钮关闭它以后, 程序的主窗口就显示出来了:



当用户在主窗口的任何地方按下鼠标左键的时候，程序会显示下面的消息框：



当用户按下主窗口右上角的  按钮来关闭对话框的时候，窗口关闭之前会显示下面的消息框：



当用户关闭了这个消息框以后，程序结束运行。

11.2.7 本节习题

1. 请描述 POINT 结构。
2. WNDCLASS 结构是怎样使用的？
3. WNDCLASS 结构的 lpfnWndProc 字段的含义是什么？
4. WNDCLASS 结构的 style 字段的含义是什么？
5. WNDCLASS 结构的 hInstance 字段的含义是什么？
6. 调用 CreateWindowEx 函数的时候，是如何把窗口的外观信息传递给函数的？
7. 试举一个调用 MessageBox 函数的例子。
8. 说出两个定义 MessageBox 函数显示的消息框上的按钮的常量。
9. 说出两个定义 MessageBox 函数显示的消息框上的图标常量。
10. 说出至少三个由 WinMain 过程所做的工作。
11. 描述 WinProc 过程在例子程序中扮演的角色。
12. 例子程序中的 WinProc 过程处理了哪些消息？
13. 描述 ErrorHandler 过程在例子程序中扮演的角色。
14. 例子程序中，调用 CreateWindow 函数后，消息框是在主窗口显示前还是显示后出现的？
15. 例子程序中，由 WM_CLOSE 消息激活的消息框是在主窗口关闭前还是关闭后出现的？

11.3 动态内存分配

动态内存分配也称为堆（内存）分配（Heap Allocation），是程序设计语言提供了一种非常有用的工具，用于为创建的对象、数组和其他结构保留内存。例如，在 Java 中，类似下面的语句会导致程序为创建的 String 对象保留内存：

```
String str = new String("abcde");
```

类似地，在 C++ 中，可能会需要为一个整数数组分配内存空间，其大小来自于一个变量：

```
int size;  
cin >> size;           // 用户输入的大小  
int array[] = new int[size];
```

C/C++ 和 Java 都有内建的运行时堆管理器，用于处理程序的存储分配和存储释放请求，堆管理器通常在程序启动时请求操作系统分配一大块内存，堆管理器创建一个空闲存储块指针链表，在接到分配请求时，把一个合适的内存块标记为保留并返回指向该内存块的指针，其后在接到针对同一内存块的释放请求时，堆管理器把该内存块放回空闲存储块的指针链表中（或释放该内存块）。每次新的分配请求到达时，堆管理器都会首先扫描空闲存储块链表，查找第一个足够大的内存块以满足分配请求。

汇编语言可通过多种方式进行动态内存分配：第一种方式是通过系统调用让操作系统为其分配内存块，第二种方式是实现字节堆管理器以处理小对象的内存分配请求。本节讲述如何使用第一种方法，例子程序都是 32 位的保护模式应用程序。

使用表 11.11 中列出的 Windows API 函数请求 MS-Windows 分配不同大小的内存块，表中所有的函数都会改写通用寄存器，因此可能需要封装这些函数以保护重要的寄存器。要想了解更多关于内存管理方面的内容，请在 Platform SDK 文档中搜索“Memory Management Reference”。

表 11.11 堆相关的函数

函 数	说 明
GetProcessHeap	在 EAX 中返回程序默认堆的句柄，堆句柄是一个 32 位的整数。如果函数调用成功，通过 EAX 中返回默认堆的句柄，否则在 EAX 中返回 0
HeapAlloc	从堆中分配一块内存。如果函数调用成功，通过 EAX 返回内存块的地址，否则在 EAX 中返回 0
HeapCreate	创建一个新的堆。如果函数调用成功，通过 EAX 返回新创建的堆的句柄，否则 EAX 中返回 0
HeapDestroy	销毁堆对象，使其句柄无效。如果函数调用成功，EAX 中返回非零值
HeapFree	释放以前从堆中分配的内存块，内存块是以堆的句柄和内存块的地址标识的。如果内存块成功释放，则返回值非 0
HeapReAlloc	调整堆中内存块的大小，必要时重新进行分配。如果函数调用成功，返回值是一个重新分配的内存块的指针，如果调用失败并且未指定 HEAP_GENERATE_EXCEPTIONS，返回值是 NULL
HeapSize	返回调用 HeapAlloc 或 HeapReAlloc 分配的内存块的大小。如果调用成功，通过 EAX 返回内存块的大小，单位是字节；如果调用失败，返回值是 SIZE_T - 1（SIZE_T 等于指针所指的值最多可能包含的字节数）

GetProcessHeap: 如果在使用程序当前拥有的默认堆满意, 可以使用 `GetProcessHeap` 函数, 该函数无参数, 在 `EAX` 中返回默认堆的句柄。函数原型如下:

`GetProcessHeap` `PROTO`

调用示例:

```
.data
hHeap HANDLE ?
.code
INVOKE GetProcessHeap
.IF eax == NULL                ; 不能获取句柄
    jmp quit
.ELSE
    mov hHeap,eax              ; 成功获取句柄
.ENDIF
```

HeapCreate: `HeapCreate` 允许为当前程序创建一个新的私有堆。函数原型如下:

```
HeapCreate PROTO,
    flOptions:DWORD,          堆分配选项
    dwInitialSize:DWORD,      堆的初始大小, 以字节为单位 .es
    dwMaximumSize:DWORD      堆的最大尺寸值, 以字节为单位 .es
```

调用时把 `flOptions` 设为 `NULL`, 把 `dwInitialSize` 设置为堆的初始大小, 实际的初始大小是该值按页边界向上舍入后的值。当调用 `HeapAlloc` 分配内存块时, 如果堆的大小超过堆的初始大小, 堆将自动增长, 上限是 `dwMaximumSize` 参数 (按页边界向上舍入) 指定的值。在调用该函数之后, 如果堆未成功创建, 则在 `EAX` 中返回 `NULL`。下面是调用示例:

```
HEAP_START = 2000000          ; 2 MB
HEAP_MAX = 400000000          ; 400 MB
.data
hHeap HANDLE ?                ; 堆的句柄
.code
INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX
.IF eax == NULL                ; 堆未创建
    call WriteWindowsMsg       ; 显示错误信息
    jmp quit
.ELSE
    mov hHeap,eax              ; 成功获取句柄
.ENDIF
```

HeapDestroy: `HeapDestroy` 销毁一个现存的私有堆 (通过调用 `HeapCreate` 创建的)。调用时传递要销毁的堆的句柄:

```
HeapDestroy PROTO,
    hHeap:DWORD                ; 堆的句柄
```

如果销毁堆失败, 则 `EAX` 中返回 `NULL`。下面是 `HeapDestroy` 的调用示例, 使用了 11.1.4 节描述的 `WriteWindowsMsg` 过程:

```
.data
hHeap HANDLE ?                ; 堆的句柄
.code
INVOKE HeapDestroy, hHeap
.IF eax == NULL
    call WriteWindowsMsg       ; 显示错误消息
.ENDIF
```

HeapAlloc: `HeapAlloc` 从堆中分配一块内存。函数原型如下:

```
HeapAlloc PROTO,
    hHeap:HANDLE,              ; 堆的句柄
```

```
dwFlags:DWORD,           ; 堆分配控制标志
dwBytes:DWORD            ; 要分配的字节数
```

调用时传递下面的参数:

- hHeap 是通过调用 GetProcessHeap 或 HeapCreate 获取的 32 位堆句柄。
- dwFlags 是包含一个或多个标志值的双字, 可以把该值设为 HEAP_ZERO_MEMORY, 此时分配的内存块将以 0 初始化。
- dwBytes 是表示要分配的内存块大小的双字, 大小是以字节为单位计算的。

如果调用成功, EAX 中返回分配的内存块的指针; 如果调用失败, EAX 中返回 NULL。下面的代码从 hHeap 标识的堆中分配 1000 个字节并以 0 初始化:

```
.data
hHeap HANDLE ?           ; 堆句柄
pArray DWORD ?          ; 指向数组的指针
.code
INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, 1000
.IF eax == NULL
    mWrite "HeapAlloc failed"
    jmp quit
.ELSE
    mov pArray, eax
.ENDIF
```

HeapFree: HeapFree 释放以前从堆中分配的内存块, 内存块是以堆句柄和内存块的地址标识的:

```
HeapFree PROTO,
    hHeap:HANDLE,
    dwFlags:DWORD,
    lpMem:DWORD
```

第一个参数是包含要释放内存块的堆的句柄, 第二个参数通常是 0, 第三个参数是指向要释放内存块的指针。如果内存块成功释放, 返回非 0 值; 如果释放失败, 则返回 0。下面是调用示例:

```
INVOKE HeapFree, hHeap, 0, pArray
```

错误处理: 如果调用 HeapCreate, HeapDestroy, GetProcessHeap 时出错, 可以调用 GetLastError API 函数或调用 Irvine32 库中的 WriteWindowsMsg 函数获取出错的细节。下面的代码调用了 Heap Create 函数, 其中包含了错误处理代码:

```
INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX
.IF eax == NULL           ; 失败了?
    call WriteWindowsMsg ; 显示错误消息
.ELSE
    mov hHeap, eax        ; 成功
.ENDIF
```

另一方面, 当 HeapAlloc 函数执行失败时, 它不设置系统错误代码, 因此不能调用 GetLastError 或 WriteWindowsMsg。

11.3.1 堆测试程序

下面的例子 (HeapTest1.asm) 使用动态内存分配的方法创建了一个 1000 字节的数组, 使用的是进程的默认堆:

```

Title Heap Test #1                                (HeapTest1.asm)
INCLUDE Irvine32.inc

; This program uses dynamic memory allocation to allocate and
; fill an array of bytes.

.data
ARRAY_SIZE = 1000
FILL_VAL EQU 0FFh

hHeap  HANDLE ?           ; 进程堆的句柄
pArray DWORD ?           ; 内存块的指针
newHeap DWORD ?          ; 新堆的句柄
str1 BYTE "Heap size is: ",0

.code
main PROC
    INVOKE GetProcessHeap           ; 获取程序默认堆的句柄
    .IF eax == NULL                ; 失败了?
        call WriteWindowsMsg
        jmp quit
    .ELSE
        mov hHeap,eax              ; 成功
    .ENDIF

    call allocate_array
    jnc arrayOk                    ; 失败了 (CF=1) ?
    call WriteWindowsMsg
    call Crlf
    jmp quit

arrayOk:                            ; 成功, 可以填充数组了
    call fill_array
    call display_array
    call Crlf

    ; free the array
    INVOKE HeapFree, hHeap, 0, pArray

quit:
    exit
main ENDP

;-----
allocate_array PROC USES eax
;
; Dynamically allocates space for the array.
; Receives: nothing
; Returns: CF = 0 if allocation succeeds.
;-----
    INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, ARRAY_SIZE

    .IF eax == NULL
        stc                        ; 返回时 CF=1
    .ELSE
        mov pArray,eax            ; 保存指针
        ctc                        ; 返回时 CF=0
    .ENDIF

    ret
allocate_array ENDP

;-----
fill_array PROC USES ecx edx esi
;
; Fills all array positions with a single character.
; Receives: nothing
; Returns: nothing
;-----

```

```

        mov     ecx,ARRAY_SIZE           ; 循环计数器
        mov     esi,pArray              ; 指向数组
L1:     mov     BYTE PTR [esi],FILL_VAL  ; 填充每个字节
        inc     esi                      ; 下一个字节
        loop    L1

        ret
fill_array ENDP
;-----
display_array PROC USES eax ebx ecx esi
;
; Displays the array
; Receives: nothing
; Returns: nothing
;-----
        mov     ecx,ARRAY_SIZE           ; 循环计数器
        mov     esi,pArray              ; 指向数组
L1:     mov     al,[esi]                 ; 取一个字节
        mov     ebx,TYPE BYTE           ; 显示之
        call    WriteHexB
        inc     esi                      ; 下一个字节
        loop    L1

        ret
display_array ENDP
END main

```

下面的例子(Heaptest2.asm)使用动态内存分配的方法循环分配 2000 个大约 0.5 MB 的内存块:

```

Title Heap Test #2                      (Heaptest2.asm)
INCLUDE Irvine32.inc

; Creates a heap and allocates multiple memory blocks,
; expanding the heap until it fails.

.data
HEAP_START = 2000000                    ; 2 MB
HEAP_MAX   = 400000000                  ; 400 MB
BLOCK_SIZE = 500000                     ; .5 MB
hHeap HANDLE ?                          ; 堆的句柄
pData DWORD ?                           ; 内存块指针
str1 BYTE 0dh,0ah,"Memory allocation failed",0dh,0ah,0

.code
main PROC
    INVOKE HeapCreate, 0,HEAP_START, HEAP_MAX
    .IF eax == NULL                      ; 失败了?
    call WriteWindowsMsg
    call CrLf
    jmp quit
    .ELSE
    mov hHeap,eax                        ; 成功
    .ENDIF

    mov ecx,2000                         ; 循环计数器
L1: call allocate_block                  ; 分配一块内存
    .IF Carry?                           ; 失败了?
    mov edx,OFFSET str1                  ; 显示出错消息
    call WriteString
    jmp quit
    .ELSE                                ; 否: 打印一个点

```

```

mov     al, '.'                ; 显示进度
call    WriteChar
.ENDIF

; call free_block              ; 可以注释/反注释该行观察效果
loop    L1

quit:
    INVOKE HeapDestroy, hHeap  ; 销毁堆
    .IF eax == NULL            ; 失败了?
    call  WriteWindowsMsg      ; 是: 显示错误消息
    call  Crlf
    .ENDIF

    exit
main ENDP

allocate_block PROC USES ecx
    ; 分配一块内存并以 0 填充
    INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, BLOCK_SIZE

    .IF eax == NULL
        stc                    ; 返回时 CF=1
    .ELSE
        mov  pData, eax         ; 保存指针
        ctc                    ; 返回时 CF=0
    .ENDIF

    ret
allocate_block ENDP

free_block PROC USES ecx
    INVOKE HeapFree, hHeap, 0, pData
    ret
free_block ENDP
END main

```

11.3.2 本节习题

1. 在 C/C++ 和 Java 中, 堆内存分配又称为什么?
2. 解释 GetProcessHeap 函数的用途。
3. 解释 HeapAlloc 函数的用途。
4. 给出一个调用 HeapCreate 函数的例子。
5. 在调用 HeapDestroy 函数时, 如何标识要销毁的内存块?

11.4 IA-32 内存管理

在 Windows 3.0 首次发布的时候, 从实模式转换到保护模式是程序员们很感兴趣的事情 (在 Windows 2.x 下面写过程序的人都会记得实模式下的 640 KB 内存限制是一件多么麻烦的事情)。随着 Windows 保护模式 (接下来是虚拟内存模式) 的到来, 全新的可能性出现了, Intel 386 处理器 (IA-32 系列处理器里面的第一种) 使这一切成为可能。在操作系统方面, 我们现在看到的是, 从不稳定的 Windows 3.0 到今天流行的、久经考验 (并且稳定) 的 Windows 及 Linux 操作系统经过了十多年的逐步演变。

本节的内容主要集中在内存管理的两个主要方面:

- 从逻辑地址到线性地址的转换。
- 从线性地址到物理地址的转换 (分页)。

让我们简要回顾一下在第 2 章中介绍的关于 IA-32 内存管理的几个术语，它们是：

- 多任务——允许同时运行多个程序（或任务）。处理器把时间分片划分并分配给每个运行中的程序。
- 段——是一块供程序存放代码或者数据的长度不定的内存。
- 分段——把多个内存段互相隔离的方法，这使多个程序可以互相隔离地运行而不会干扰。
- 段描述符——是用来描述一个内存段的 64 位值，其中包含了段的基地址、访问权限、长度限制、段的类型和使用方式等信息。

现在加入两个新的概念：

- 段选择子——存放在段寄存器（CS，DS，SS，ES，FS 和 GS）里面的 16 位值。
- 逻辑地址——一个段选择子和一个 32 位的偏移地址的组合。

本书几乎所有的内容都忽略了段寄存器而只用到了 32 位的数据偏移地址，因为用户程序从不直接修改段寄存器。不过，从系统程序员的视角来看，段寄存器是很重要的，因为它间接地和内存中的段相关。

11.4.1 线性地址

逻辑地址到线性地址的转换

多任务操作系统允许多个程序（任务）同时在内存中运行，每个程序拥有属于它自己的唯一的数据空间。假设有三个程序，每个程序都在偏移地址 200h 处有一个变量，这三个变量是如何互相隔离的呢？答案是：IA-32 处理器使用了一个经过一步或者两个步骤的过程把每个变量的地址转换到另一个唯一的偏移地址上去。

第一步是把变量的段和偏移地址合成一个线性地址，线性地址有可能就是变量的物理地址，但是有些操作系统（如 Windows 或者 Linux）使用一种称为 IA-32 的分页技术，使程序能够使用比计算机中实际物理内存更多的线性地址空间。如果情况是这样，就要经过第二个步骤，使用页面转换的方法把线性地址转换到物理地址。有关页面转换的内容将在 11.4.2 节介绍。

首先，我们来看看处理器是如何使用段和偏移地址来确定一个变量的线性地址的。每个段选择子指向描述符表里面的一个段描述符，段描述符包含了段的基址（起始地址），如图 11.6 所示，逻辑地址中的 32 位偏移地址和段的基址相加就得到了线性地址。

线性地址：线性地址是一个介于 0 到 FFFFFFFh 的 32 位整数，它代表内存中的一个位置。如果分页机制没有打开的话，那么线性地址实际上就是数据的物理地址。

分页

分页机制是 IA-32 系列处理器的一个重要特征，它使得计算机同时在内存中运行原本无法装入的一堆程序成为可能。在一开始，处理器仅仅装入程序的一部分，剩余的部分保留在磁盘上面。程序要用到的内存被划分成称为页的小块，通常每块的大小为 4 KB。运行每个程序的时候，处理器有选择地在内存中释放一些不用的页面，然后装入其他马上要被用到的页面。

操作系统使用一个页目录和一系列的页表来追踪内存中所有程序的页面使用情况。当一个程序尝试访问线性地址空间中的某个地址的时候，处理器自动把线性地址转换成物理地址，这个转换就称为页面转换。如果需要的页面尚未在内存中，处理器打断程序的执行并引发一个页错误，操作系统捕获这个错误并在程序恢复运行前把所需的页面从磁盘复制到内存中。从应用程序的角度来看，页错误和页面转换是自动发生的。

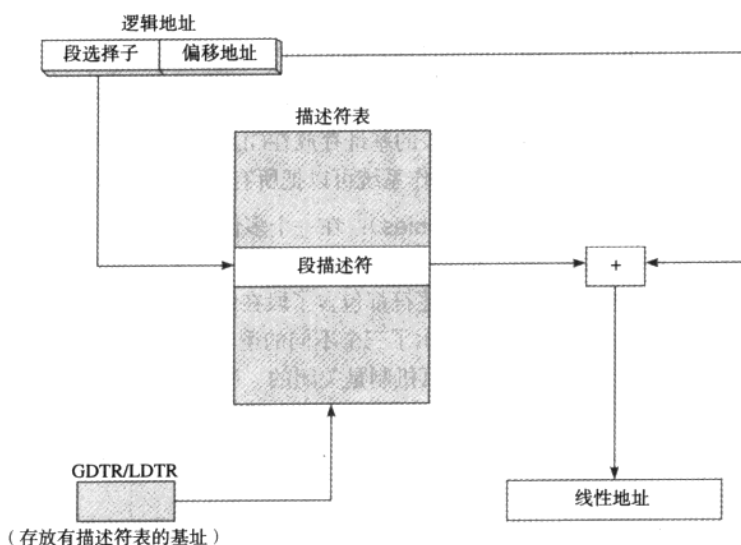


图 11.6 逻辑地址到线性地址的转换

举例来说，读者可以在 Windows 2000 中打开任务管理器程序并看看其中显示的物理内存和虚拟内存之间的差别。图 11.7 显示了一个装有 256 MB 物理内存的计算机的情况。当前使用中的虚拟内存的总数量显示在任务管理器的 Commit Charge 一栏中，请注意图中显示的最大可用虚拟内存为 633 MB，明显大于计算机的物理内存数量。

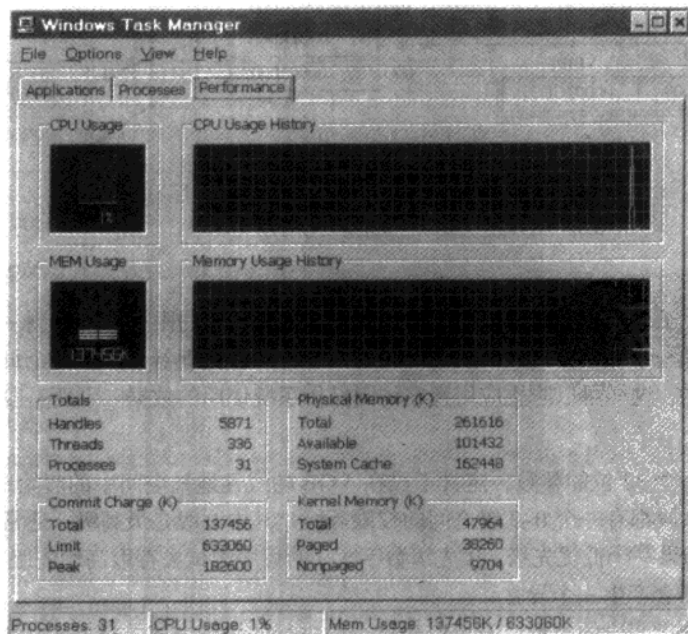


图 11.7 Windows 任务管理器的例子

描述符表

段描述符存在于两种类型的表中：全局描述符表 (GDT) 和局部描述符表 (LDT)。

全局描述符表 (GDT, Global Descriptor Table)：系统中只存在一个全局描述符表，系统在处理器切换到保护模式时创建全局描述符表，表的基址存放在 GDTR (全局描述符表寄存器) 里面。表中的项目 (称为段描述符) 指向各个段。操作系统可以把所有程序都要使用的段存放在 GDT 中。

局部描述符表 (LDT, Local Descriptor Tables)：在一个多任务的操作系统中，每个程序或任务都有它自己的段描述符表，这个表称为局部描述符表 (LDT)。当前程序的 LDT 的基址存放在 LDTR (局部描述符表寄存器) 中。每个段描述符都包含了段在线性地址空间中的基址。如图 11.8 所示，一个段和其他段通常是不同的。图中显示了三个不同的逻辑地址，每个地址分别对应于 LDT 中的不同表项。在这个例子中，我们假设分页机制是关闭的，所以线性地址空间也就是物理地址空间。

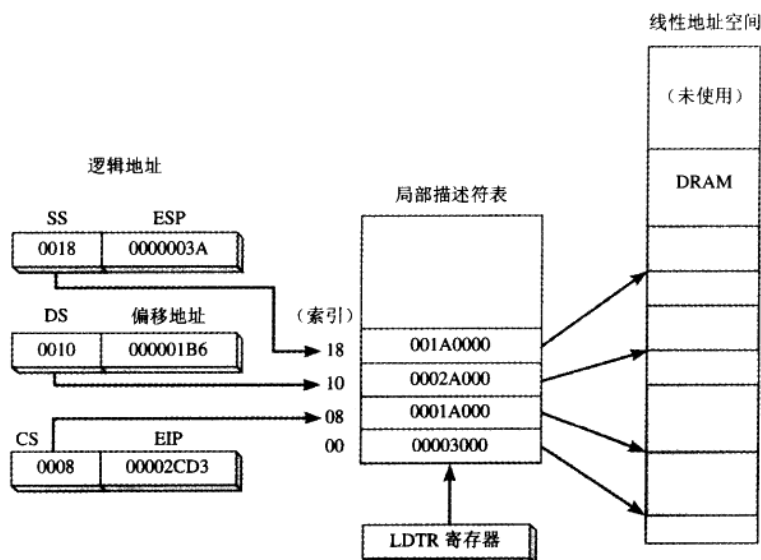


图 11.8 局部描述符表中的索引

段描述符的细节

段描述符中除了包含段的基址以外，有些数据位定义了段的限长和段类型。代码段是一个只读段的例子，如果一个程序尝试去修改代码段的内容，那么处理器会产生一个页异常。段描述符中也包含保护级别，这样可以防止应用程序访问操作系统使用的数据。下面是段描述符中各个域的含义。

基地址：是一个 32 位的整数，定义了段在 4 GB 的线性地址空间中的起始地址。

特权级：每个段都有一个 0~3 级之间的权限等级，其中 0 级是最高级，通常被操作系统的核心代码所使用。如果一个低优先级 (优先级数字大) 的程序尝试去存取高优先级 (优先级数字小) 的段，那么处理器会产生一个异常。

段类型：用来指明段的类型以及可以对这个段进行的访问方式，还有段的扩展方向 (向上或向下)。数据段 (包括堆栈段) 可以是只读或者是可读写的，可以向上或向下扩展。代码段可以仅仅是可执行的或者是可执行/可读的。

段存在标志：这个数据位指明段当前是否在物理内存中存在。

粒度标志：用来决定如何解释段限长域的数值，如果标志位清零，那么段限长的单位是字节。如果该标志位置位，那么段限长的单位是 4096 字节。

段限长：是一个 20 位的整数，表示段的长度，它根据粒度标志的值按下面的两种方式解释：

- 1 字节到 1 MB 字节的段长度。
- 4096 字节到 4 GB 字节的段长度。

11.4.2 页面地址转换

当分页机制被允许的时候，处理器必须把 32 位的线性地址转换到 32 位的物理地址^①，在这个过程中要用到以下三个数据结构：

- 页目录：一个最多包含 1024 个 32 位表项的页表地址表。
- 页表：一个最多包含 1024 个 32 位表项的页地址表。
- 页：一个 4 KB 或者 4 MB 的地址空间。

为了简单起见，在接下来的讨论中假设使用 4 KB 的页。

一个线性地址可以被划分为三个部分：指向页目录的指针、指向页表的指针和在页中的偏移地址。页目录的起始地址存放在控制寄存器 (CR3) 中。如图 11.9 所示，当线性地址被转换到物理地址的时候，处理器执行了以下的步骤。

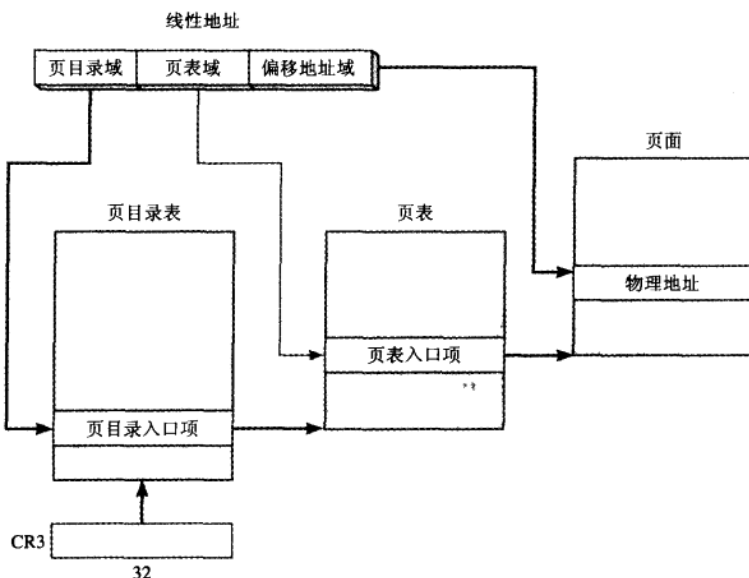


图 11.9 线性地址到物理地址的转换

1. 线性地址代表线性地址空间中的一个位置。
2. 以线性地址中 10 位的页目录域作为索引，从页目录表中得到页表入口项，页表入口项中包含了页表的基址。

① 奔腾 Pro 以及后续的处理器的允许用 36 位的地址，这一点不在本书的讨论范围之内。

3. 以线性地址中 10 位的页表域作为索引, 从页表入口项指定的表项中得到页在物理内存中的基址。
4. 线性地址中 12 位的偏移地址域加上页的基址, 就得到了操作数确切的物理地址。

操作系统可以选择让所有的程序或任务使用同一个页目录, 或者让每个任务使用单独的页目录, 也可以混合使用两种方式。

MS-Windows 的虚拟机管理器

现在我们已经有了 IA-32 处理器如何管理内存的总体概念了, Windows 又是如何管理内存的呢? 这也应该是个令人感兴趣的问题。下面的小段摘自 Microsoft Platform SDK 文档。

虚拟机管理器(VMM, Virtual Machine Manager)是位于 MS-Windows 核心的 32 位保护模式操作系统, 它的主要职责是创建、运行、监控和终止虚拟机。虚拟机提供了内存管理、进程、中断和异常等服务, 它和虚拟设备(32 位的保护模式模块)协同工作, 使虚拟设备能够以截取中断和异常的方式来控制应用程序对硬件和所安装软件的操作。不管是 VMM 还是虚拟设备, 都运行在特权级 0 下的同一个 32 位平坦模式的地址空间中, 系统在全局描述符中创建了两个入口(段描述符), 一个给代码段, 另一个给数据段。两个段的基址都是从线性地址 0 开始的, 并且永远不会被改变。VMM 支持多线程和抢先式的多任务机制, 它在多个虚拟机之间共享 CPU 时间, 这样在这些虚拟机之中运行的应用程序就能够同时运行。

在上面一段话中, 我们可以把虚拟机解释为 Intel 称之为“进程”或者“任务”的东西, 它由程序代码、支持软件、内存和寄存器等组成。每个虚拟机都有属于它自己的地址空间、I/O 地址空间、中断向量表和局部描述符表。在虚拟 8086 模式下运行的应用程序在特权级 3 下运行。在 MS-Windows 中, 保护模式程序可以在特权级 0 和特权级 3 下面运行(特权级 1 和 2 在 Windows 下未使用)。

11.4.3 本节习题

1. 名词解释: 多任务, 分段。
2. 名词解释: 段选择子, 逻辑地址。
3. (对/错) 一个段选择子指向段描述符表中的一个表项。
4. (对/错) 段描述符包含了段的基址。
5. (对/错) 段选择子是 32 位长的。
6. (对/错) 段描述符不包含段的限长信息。
7. 试描述一下线性地址。
8. 分页和线性内存地址之间有什么关系?
9. 如果分页机制被关闭, 那么处理器如何把线性地址转换成物理地址?
10. 分页有什么优点?
11. 哪个寄存器中存放有局部描述符表的基址?
12. 哪个寄存器中存放有全局描述符表的基址?
13. 系统中可以存在多少个全局描述符表?
14. 系统中可以存在多少个局部描述符表?
15. 请举出至少 4 个段描述符中的字段名称。

16. 分页转换过程中涉及到哪些数据结构?
17. 哪个数据结构中包含了页表的基址?
18. 哪个数据结构中包含了页的基址?

11.5 本章小结

从表面看, 32 位的控制台程序和 16 位的 MS-DOS 应用程序在外观和行为上都是很相似的, 它们都使用标准的输入输出设备, 都支持命令行的重定向操作, 也都可以输出彩色的文本。但实质上, 32 位控制台程序和 MS-DOS 程序却是完全不同的, 前者在保护模式下运行, 而后者在实模式下运行。另外, 它们使用的也是完全不同的函数库, Win32 控制台程序使用的就是 Windows 图形界面程序使用的那些库文件, 而 MS-DOS 程序被限制于使用 BIOS 和 MS-DOS 中断, 这些中断在 IBM-PC 的那个年代就已经在使用了。

在 Win32 API 中可以使用两种字符集: 8 位的 ASCII/ANSI 字符集和 16 位的宽字符/Unicode 字符集。

写汇编程序的时候, API 函数中使用的标准 Windows 数据类型必须先转换成 MASM 数据类型 (参见表 11.1)。

控制台句柄是一个用于控制台输入输出的 32 位整数。GetStdHandle 函数用来得到控制台句柄。高级的控制台输入使用 ReadConsole 函数, 高级的控制台输出使用 WriteConsole 函数。我们使用 CreateFile 函数来创建或者打开一个文件, 用 ReadFile 函数读取文件, 用 WriteFile 函数写文件, 并且使用 CloseHandle 函数来关闭文件。如果要移动文件读写指针, 可以使用 SetFilePointer 函数。

SetConsoleScreenBufferSize 用来控制控制台屏幕缓冲区。SetConsoleTextAttribute 函数用来改变文本颜色。本章中的 WriteColors 例子程序演示了 WriteConsoleOutputAttribute 函数和 WriteConsoleOutputCharacter 函数的用法。

GetLocalTime 函数可以用来获取系统时间, SetLocalTime 用来设置系统时间, 两个函数都用到了 SYSTEMTIME 结构。本章例子中的 GetDateTime 例子程序返回一个以 64 位整数表示的系统时间, 这个数值是自 1601 年 1 月 1 日开始的以 100 ns 为单位的计数值。

当要创建一个图形界面的 Windows 应用程序的时候, 我们需要填写包含主窗口的窗口类信息的 WNDCLASS 结构, 还必须创建一个 WinMain 过程来获取当前程序的句柄, 并加载图标和鼠标光标, 然后注册窗口类, 创建主窗口, 接下来显示并更新主窗口, 最后, 我们开始一个消息循环来接收并分派消息。

WinProc 过程负责接收并处理输入的窗口消息, 这些消息通常由按动鼠标或按下键盘等用户动作而激发。本章中的例子程序处理 WM_LBUTTONDOWN 消息、WM_CREATE 消息和 WM_CLOSE 消息, 当检测到这些消息的时候, 程序会显示一个消息框。

动态内存分配, 也称为堆 (内存) 分配 (Heap Allocation), 是程序用于保留和释放内存的有用工具。汇编语言可通过多种方式进行动态内存分配: 第一种方式是通过系统调用让操作系统为其分配内存块; 第二种方式是实现自己的堆管理器以处理小对象的内存分配请求。下面是一些用于动态内存分配的最重要的 Win32 API 调用:

- GetProcessHeap 返回程序默认堆的句柄, 该句柄是一个 32 位整数值。
- HeapAlloc 从堆中分配一块内存。
- HeapCreate 创建一个新的堆。

- HeapDestroy 销毁一个堆。
- HeapFree 释放以前从堆中分配的内存块。
- HeapReAlloc 调整堆中内存块的大小, 必要时重新进行分配。
- HeapSize 返回以前分配的内存块的大小。

本章的内存管理一节集中讨论两个主题: 逻辑地址到线性地址的转换和线性地址到物理地址的转换。

逻辑地址中的选择子部分指向段描述符表中的一个表项, 这个表项指向线性内存中的一个段。段描述符中包含了这个段的相关信息, 如界限、访问类型等。系统中有两种描述符表: 一个全局描述符表 (GDT) 和一个或多个局部描述符表 (LDT)。

分页是 IA-32 系列处理器的一个重要特征, 它使得计算机同时在内存中运行原本无法装入的一堆程序成为可能。在一开始, 处理器仅仅装入程序的一部分, 剩余的部分保留在磁盘上面。处理器使用页目录、页表和页得到数据的物理地址。一个页目录表包含了指向各个页表的指针, 而一个页表包含了指向多个页的指针。

阅读材料: 下面的书目对进一步了解 Windows 编程有很大帮助:

- Mark Russinovich 和 David Solomon, *Microsoft Windows Internals*, 第四版, Microsoft 出版社, 2004。
- Barry Kauler, *Windows Assembly Language and System Programming*, CMP Books, 1997。
- Charles Petzold, *Programming Windows*, 第五版, Microsoft 出版社, 1998。

11.6 编程练习

1. ReadString

使用堆栈传递参数的方式书写你自己的 ReadString 过程, 向它传递两个参数: 一个是指向字符串缓冲区的指针, 一个是代表最多可输入字符数的整数。函数要在 EAX 中返回实际输入的字符数量。过程必须从控制台输入一个字符串, 并在字符串的尾部添上 0 字符 (在 0Dh 字符处的位置)。可以在 11.1.4 节查阅 Win32 函数 ReadConsole 的细节。最后写一个小程序来测试这个过程。

2. 字符串输入输出

使用 Win32 函数 ReadConsole 写一个程序, 从用户处输入以下信息: 姓名、年龄、电话号码, 然后把这些信息加上标号用 WriteConsole 函数以美观的格式显示出来。请不要使用 Irvine32 库函数里面的任何过程。

3. 清除屏幕

Irvine32 库中有个 Clrscr 过程, 可以用来清屏, 请自己编写另外一个版本的 Clrscr 过程。

4. 随机填写屏幕

写一个用随机的颜色和随机的字符填写屏幕上每个位置的程序。另外, 如果要求字符的颜色有 50% 的可能性是红色时程序应如何编写?

5. 画框

用本书前言部分字符集中的画线字符在屏幕上画一个方框。提示: 使用 WriteConsoleOutputCharacter 函数。

6. 学生记录

写一个程序完成下列功能：首先创建一个新的文本文件，然后提示用户输入学生的身份证号、姓名和生日，并把这些信息写入文件。用同样的方式输入多条记录后关闭文件。

7. 卷动文本窗口

写一个程序完成下列功能：在控制台屏幕缓冲区中写 50 行文本，每行的文本内容中要有行号。把控制台窗口移动到屏幕缓冲区的顶部，然后以每秒 2 行的稳定速度向下卷动文本，直到控制台窗口到达屏幕缓冲区的底部为止。

8. 方块动画

用几个 ASCII 码 DBh 在屏幕上画一个彩色的小方块，然后每隔固定的 50 ms 的时间在屏幕上把方块向一个随机的方向移动。另外：如果要求间隔时间在 10~100 ms 之间随机产生时程序该如何编写？

9. 文件的最后访问时间

写一个过程 LastAccessDate，以文件时间和日期戳记填充 SYSTEMTIME 结构，要求通过 EDX 传递文件名，通过 ESI 传递 SYSTEMTIME 结构的偏移地址。如果函数无法找到文件，则设置进位标志。在实现该过程时，需要打开文件并获取文件的句柄，然后把句柄传递给 GetFileTime 函数，接下来把 GetFileTime 函数的输出传递给 FileTimeToSystemTime 函数，最后关闭文件。写一个测试程序调用该过程并显示一个特定文件最后被访问的时间，输出示例如下：

```
ch11_09.asm was last accessed on: 6/16/2005
```

10. 读取大文件

修改 11.1.8 节的 ReadFile.asm 程序，以使其能够读取大于输入缓冲区的文件。把缓冲区的大小减少到 1024 字节，使用一个循环持续读取并显示文件的内容，直到无法读到更多的数据为止。如果要使用 WriteString 显示缓冲区内容，一定要记得在缓冲区的末尾添加一个 0。

11. 链表

（高级）使用本章讲述的动态内存分配函数实现一个单链表，节点名为 Node（参加第 10 章），其中包含一个整数和一个指向链表中下一个节点的指针。通过一个循环要求用户输入一些整数值。对于输入每个整数，都分配一个 Node 对象，把整数插入 Node 对象，然后再把 Node 对象追加到链表的结尾。如果输入了整数 0，则终止循环。最后，从头到尾重新显示链表（只有曾经使用高级语言创建过链表的读者才应尝试这个题目）。

第 12 章 高级语言接口

本章要点

- 简介
- 内联汇编代码
- 在保护模式下与 C/C++ 链接
- 在实地址模式下与 C/C++ 链接

12.1 简介

大多数程序员并不使用汇编语言编写大规模的程序，原因很简单，因为用纯汇编编写程序太耗时了。高级语言隐藏了大量的细节信息，其目的是想把程序员从大量的细节中解放出来，以加快项目的进度。不过汇编语言仍然被广泛使用着，主要用于配置硬件设备以及优化程序的执行速度和尺寸大小等方面。

本章着重讲述汇编语言和高级语言之间的接口（interface）或者连接（connection）。第一节讲述如何在 C++ 中编写内联汇编代码，接下来的一节讲述如何把独立的汇编语言模块同 C++ 程序链接在一起，并针对保护模式程序和实地址模式程序分别给出了例子。在本章的最后讲述如何在汇编语言中调用 C/C++ 函数。

12.1.1 约定

从高级语言中调用汇编语言过程需要着重考虑以下一些因素。

首先，一种语言使用的命名约定是指语言中过程及变量命名的规则或特性。例如，我们不得不回答一个重要的问题：编译器改变目标文件中的标识符名吗？如果改变的话，它又是如何改变的呢？

其次，汇编语言中的段名必须同高级语言使用的段名兼容。

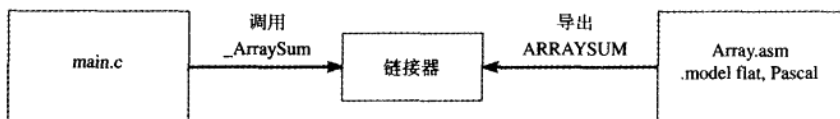
再次，程序使用的内存模式（微型、小型、紧凑、中型、大型、巨型或平坦）决定了段的尺寸（16 位或 32 位）、调用和引用是近（near，在同一个段内）还是远（far，在不同的段中）。

调用约定：调用约定是指过程如何被调用的底层细节。以下的细节是必须考虑的：

- 被调用过程必须保护哪些寄存器。
- 传递参数的方法——通过寄存器、堆栈还是共享内存区传递参数，或者使用其他方法。
- 调用程序向过程传递参数的顺序。
- 参数是以传值方式还是以传引用方式进行传递的。
- 过程调用之后堆栈指针是如何恢复的。
- 过程如何向调用程序返回结果。

外部标识符：从其他语言编写的程序中调用汇编语言过程时，外部标识符的命名约定必须兼容。外部标识符（external identifier）是放置在模块的目标文件中、链接器可使其对其他程序模块而言可见的那些名字。链接器可以处理对外部标识符的引用，但只有在命名约定一致的情况下链接器才能处理。

例如, 假设名为 `main.c` 的 C 程序调用了名为 `ArraySum` 的外部过程, 如下图所示, C 编译器自动保留了过程名的大小写并在前面加了一条下划线, 把它变成了 `_ArraySum`:



而 `Array.asm` 模块是用汇编语言编写的, 由于 `.MODEL` 伪指令使用了 Pascal 语言选项, `ArraySum` 过程被导出为 `ARRAYSUM`。由于两个导出名字不同, 因此链接器无法生成可执行文件。

较老的程序设计语言编译器 (如 COBOL 和 Pascal) 通常把名字全部转换成大写字母, 比较新的语言 (如 C, C++ 和 Java) 则保留标识符的大小写。此外, 一些支持函数重载的语言 (如 C++) 使用一种称为名字修饰的技术在函数名后面添加其他字符, 例如函数 `MySub(int n, double b)` 可能被导出为 `MySub#int#double`。

在汇编语言模块中, 可在 `.MODEL` 伪指令 (细节参见 8.4.1 节) 中通过选择语言关键字来控制大小写是否敏感。

段的名称: 在把汇编语言编写的过程同高级语言编写的程序相链接时, 段的名称必须兼容。本章中我们使用 Microsoft 简化段伪指令, 如 `.CODE`, `.STACK` 和 `.DATA` 等, 因为它们与 Microsoft C++ 编译器生成的段名是兼容的。

内存模式: 调用程序和被调用过程必须使用同样的内存模式。在实地址模式下, 可选择小型 (small)、紧凑 (compact)、中型 (medium)、大型 (large) 和巨型 (huge) 模式; 在保护模式下则必须使用平坦 (flat) 模式。本章就两种模式分别举了例子。

12.1.2 本节习题

1. 程序语言使用的命名约定的含义是什么?
2. 实地址模式下可用的内存模式有哪些?
3. 使用 Pascal 语言关键字的汇编语言过程可以同 C++ 程序相链接吗?
4. 高级语言程序调用汇编语言过程的时候, 调用程序和过程必须使用同样的内存模式吗?
5. 为什么在从 C 和 C++ 程序中调用汇编语言过程时保持大小写敏感是很重要的?
6. 一种语言的调用约定是否包括了保护过程使用的特定寄存器?

12.2 内联汇编代码

12.2.1 Microsoft Visual C++ 中的 `_asm` 伪指令

内联汇编代码是直接插入到高级语言程序中的汇编语言源代码, 大多数现代 (C/C++) 编译器都支持内联汇编, 如 Borland C++, Pascal 和 Delphi 编译器等。

本节讲述如何使用 Microsoft Visual C++ 为运行于 32 位保护模式下并使用平坦内存模式的程序编写内联汇编代码, 其他支持内联汇编的高级语言编译器使用的格式可能有所不同。

与在外部模块中编写汇编代码相比, 内联汇编代码显得更加直接, 编写内联汇编代码的主要优点是它的简单性, 因为程序员不必考虑外部链接、命名以及参数传递协议等问题。

使用内联汇编代码的最主要缺点是缺乏可移植性, 当高级语言程序必须为多种目标平台分别编译的时候, 这就成了一个严重的问题。例如, 运行于奔腾处理器上的内联汇编代码不能在 RISC

处理器上运行。某种程度上,在源代码中插入条件编译指令为不同的目标平台编写不同版本的函数可解决部分问题,但显而易见,维护仍将是个大问题。与之相反,外部汇编语言模块则很容易用为不同目标平台设计的类似的链接库代替。

__asm 伪指令:在 Visual C++ 中, __asm 伪指令可放在汇编语句的前面或用于标记一块汇编语句(称为汇编语句块)的开始,使用格式如下:

```
-- asm statement

-- asm {
    statement-1
    statement-2
    ...
    statement-n
}
```

(注意在 asm 前面有两个下画线字符。)

注释:注释可放在汇编语句块中任何语句的后面,使用汇编语言格式的注释或 C/C++ 格式的注释都是可以的。Visual C++ 手册建议尽量避免使用汇编风格的注释,因为这样有可能同 C 宏相冲突。下面是允许的注释格式:

```
mov esi,buf      ; initialize index register
mov esi,buf      // initialize index register
mov esi,buf      /* initialize index register */
```

特性:以下是编写内联汇编代码时可以做的事情:

- 使用任何 Intel 指令集中的指令。
- 使用寄存器操作数。
- 通过名字引用函数参数。
- 可以引用在汇编语句块外面声明的代码标号和变量(这是很重要的,因为函数的局部变量必须在汇编块外声明)。
- 使用汇编风格或 C 风格的数值表示法。例如,内联汇编代码中可使用 0A26h 或 0xA26,二者是等价的。
- 在语句中使用 PTR 操作符,例如 inc BYTE PTR[esi]。
- 使用 EVEN 和 ALIGN 伪指令。

限制:以下是在编写内联汇编代码时不能使用的特性:

- 使用数据定义伪指令,如 DB(BYTE)和 DW(WORD)等。
- 使用汇编操作符(PTR 除外)。
- 使用 STRUCT, RECORD, WIDTH 和 MASK。
- 使用宏伪指令,包括 MACRO, REPT, IRC, IRP 和 ENDM 等,或宏操作符(<>, !, &, %和 TYPE)。
- 引用段名(但可使用段寄存器作为操作数)。

寄存器值:在汇编语句块的开头对寄存器的值不能做任何假设,因为在汇编语句块前面执行的语句可能已经对寄存器进行了修改。Microsoft Visual C++ 的关键字 __fastcall 导致编译器使用寄存器来传递参数。为了避免寄存器冲突,不要同时使用 __fastcall 和 __asm。

通常,内联汇编代码可以修改 EAX, EBX, ECX 和 EDX 寄存器,因为编译器并不假设语句

之间会保留这些寄存器的值。如果修改的寄存器较多，编译器就几乎不可能对同一过程内的 C++ 代码进行完全的优化了，因为优化也要使用寄存器。

虽然不能使用 OFFSET 操作符，但可以使用 LEA 指令返回变量的偏移值，例如下面的指令把 buffer 的偏移地址送到 ESI：

```
lea esi,buffer
```

长度、类型和大小：在内联汇编中可使用 LENGTH、SIZE 和 TYPE 操作符。LENGTH 返回数组中元素的数目。根据目标值的不同，TYPE 可返回下列值之一：

- C/C++ 类型或标量变量占用的字节数。
- 结构使用的字节数。
- 对数组而言，TYPE 返回单个数组元素的大小。

SIZE 操作符返回 LENGTH \times TYPE 的值。后面的程序片断说明了在内联汇编中各种 C++ 类型的返回值。

Microsoft Visual C++ 的内联汇编并不支持 MASM 6.0 引入的 SIZEOF 和 LENGTHOF 操作符。

使用 LENGTH、TYPE 和 SIZE 操作符

下面的程序包含了在内联汇编中使用 LENGTH、TYPE 和 SIZE 操作符来计算 C++ 变量相关值的代码，每个表达式的返回值都列在同一行的注释中。

```
struct Package {
    long originZip;           // 4
    long destinationZip;     // 4
    float shippingPrice;     // 4
};

char myChar;
bool myBool;
short myShort;
int myInt;
long myLong;
float myFloat;
double myDouble;
Package myPackage;

long double myLongDouble;
long myLongArray[10];

__asm {
    mov eax, myPackage.destinationZip;

    mov eax, LENGTH myInt;    // 1
    mov eax, LENGTH myLongArray; // 10

    mov eax, TYPE myChar;    // 1
    mov eax, TYPE myBool;    // 1
    mov eax, TYPE myShort;   // 2
    mov eax, TYPE myInt;     // 4
    mov eax, TYPE myLong;    // 4
    mov eax, TYPE myFloat;   // 4
    mov eax, TYPE myDouble;  // 8
```

```

mov eax,TYPE myPackage;      // 12
mov eax,TYPE myLongDouble;   // 8
mov eax,TYPE myLongArray;    // 4

mov eax,SIZE myLong;         // 4
mov eax,SIZE myPackage;      // 12
mov eax,SIZE myLongArray;    // 40
}

```

12.2.2 文件加密的例子

现在让我们编写一个简短的小程序来读文件，然后加密文件并把输出写入一个新文件中。TranslateBuffer 函数使用了一个 `_asm` 块，其中定义的内联汇编语句循环遍历字符数组并用预定义值对每个字符进行异或操作。内联汇编语句可以引用函数参数、局部变量和代码标号，由于该例是使用 Microsoft Visual C++ 编译的 Win32 控制台应用程序，所以无符号整数数据类型是 32 位的：

```

void TranslateBuffer( char * buf,
    unsigned count, unsigned char eChar )
{
    __asm {
        mov esi,buf
        mov ecx,count
        mov al,eChar

L1:
        xor [esi],al
        inc esi
        loop L1
    } // asm
}

```

C++模块：C++的启动代码从命令行中读取输入和输出文件名，接着通过一个循环从文件中读取一块数据，调用 TranslateBuffer 对读入的数据块加密并把加密后数据缓冲区写入到新文件中：

```

// ENCODE.CPP - 复制并加密一个文件
#include <iostream>
#include <fstream>
#include "translat.h"

using namespace std;

int main( int argcount, char * args[] )
{
    // 从命令行读取输入文件和输出文件名
    if( argcount < 3 ) {
        cout << "Usage: encode infile outfile" << endl;
        return -1;
    }

    const int BUFSIZE = 2000;
    char buffer[BUFSIZE];
    unsigned int count; // 字符计数

    unsigned char encryptCode;
    cout << "Encryption code [0-255]? ";
    cin >> encryptCode;

    ifstream infile( args[1], ios::binary );
    ofstream outfile( args[2], ios::binary );

    cout << "Reading" << args[1] << "and creating"
        << args[2] << endl;
}

```

```

while (!infile.eof() )
{
    infile.read(buffer, BUFSIZE);
    count = infile.gcount();
    TranslateBuffer(buffer, count, encryptCode);
    outfile.write(buffer, count);
}
return 0;
}

```

在命令行上运行这个程序是非常简单的，只需传递输入和输出文件的名字即可。例如下面的命令行读取 infile.txt 并生成 encoded.txt:

```
encode infile.txt encoded.txt
```

头文件: translate.h 头文件中包含了 TranslateBuffer 的函数原型:

```
void TranslateBuffer( char * buf, unsigned count,
                    unsigned char eChar );
```

可在本书的附带代码的\Examples\ch12\VisualCPP\Encode 目录下查看该程序。

过程调用的开销

在使用调试器调试该程序的时候，如果查看一下反汇编窗口并看看调用过程和从过程返回的开销究竟有多大是非常有趣的。下列语句在堆栈上压入三个参数并调用 TranslateBuffer。在 Visual C++中，我们激活了 Show Source Code 和 Show Symbol Names 选项:

```

; TranslateBuffer(buffer, count, encryptCode)
mov  al,byte ptr [encryptCode]
push eax
mov  ecx,dword ptr [count]
push ecx
lea  edx,[buffer]
push edx
call TranslateBuffer (41598Fh)
add  esp,0Ch

```

以下是 TranslateBuffer 的反汇编代码，注意编译器自动插入了多条语句以设置 EBP 并保护一系列寄存器。不管程序是否实际修改了这些寄存器，编译器总是对它们进行保护:

```

push  ebp
mov   ebp,esp
sub   esp,40h
push  ebx
push  esi
push  edi

; 内联代码从这里开始
mov   esi,dword ptr [buf]
mov   ecx,dword ptr [count]
mov   al,byte ptr [eChar]
L1:
    xor  byte ptr [esi],al
    inc  esi
    loop L1 (41D762h)
; 内联代码结束

pop   edi
pop   esi
pop   ebx
mov   esp,ebp

```

```
pop    ebp
ret
```

如果在 Visual C++ 的反汇编窗口中关闭 Show Symbol Names 选项, 三条参数送寄存器的指令则显示为:

```
mov     esi,dword ptr [ebp+8]
mov     ecx,dword ptr [ebp+0Ch]
mov     al,byte ptr [ebp+10h]
```

本例中我们指示编译器生成调试 (Debug) 目标, 这时生成的是适于交互调试的未经优化的目标代码, 如果选择了发布 (Release) 目标, 编译器则生成更加高效 (但更难阅读) 的代码。12.3.1 节将展示编译器生成的优化代码。

省略过程调用: 本节最开始给出的 TranslateBuffer 函数中包含的是 6 条指令, 最后事实上总共需要执行 18 条指令。如果该函数被调用了上万次, 执行时间可能就值得注意了。为了避免函数调用的开销, 可以在调用 TranslateBuffer 函数的循环中直接插入内联汇编代码, 以创建一个更加高效的程序:

```
while (!infile.eof() )
{
    infile.read(buffer, BUFSIZE );
    count = infile.gcount();
    __asm {
        lea esi,buffer
        mov ecx,count
        mov al,encryptCode
    L1:
        xor [esi],al
        inc esi
        Loop L1
    } // asm
    outfile.write(buffer, count);
}
```

可在本书的附带代码\Examples\ch12\VisualCPP\Encode_Inline 目录下查看该程序。

12.2.3 本节习题

1. 内联汇编代码与 C++ 的内联函数有什么不同?
2. 内联汇编代码与使用外部汇编语言过程相比有什么优点?
3. 列举至少两种在内联汇编语言代码中写注释的方法。
4. (是/否) 内联语句能引用 __asm 块外的代码标号吗?
5. (是/否) EVEN 和 ALIGN 语句都能在内联汇编代码中使用吗?
6. (是/否) 内联汇编代码中可以使用 OFFSET 操作符吗?
7. (是/否) 内联汇编代码中的变量能用 DW 和 DUP 操作符定义吗?
8. 使用 __fastcall 调用约定的时候, 如果内联汇编代码中修改了寄存器则会发生什么?
9. 如果不使用 OFFSET 操作符, 有其他的方法把变量偏移送变址寄存器吗?
10. 当用于 32 位整数数组时, LENGTH 操作符返回什么值?
11. 当用于 32 位长整数数组时, SIZE 操作符返回什么值?

12.3 在保护模式下与 C++ 程序链接

为 IA-32 处理器编写的保护模式程序有时会有瓶颈, 必须进行速度优化。如果目标系统是嵌入式系统, 那么可能还有严格的内存大小的限制。本节讲述在保护模式下如何用汇编语言编写可以从 C 和 C++ 程序中调用的外部过程。这一类程序至少由两个模块构成: 第一个模块是包含外部过程的汇编模块; 第二个模块是包含程序启动和结束代码的 C/C++ 模块。在编写汇编语言外部过程的时候, C/C++ 的某些特殊的要求或特性会影响编写汇编代码的方式。

参数: C/C++ 程序按照参数在参数列表表中出现的顺序从右向左传递, 过程返回后由调用程序负责清理堆栈。清理堆栈时可以给堆栈指针加上参数所占空间的大小或从堆栈中弹出足够数量的值。

外部名字: 在汇编语言模块源码中, 应该在 .MODEL 伪指令中指定 C 调用约定并且为外部 C++ 程序调用的每个过程都声明一个原型:

```
.586
.model flat, C
AsmFindArray PROTO,
    srchVal:DWORD, arrayPtr:PTR DWORD, count:DWORD
```

函数声明: 在 C 语言程序中, 应使用 extern 关键字来声明外部汇编语言过程。例如下面是 AsmFindArray 的声明:

```
extern bool AsmFindArray( long n, long array[], long count );
```

如果是在 C++ 程序中调用该过程, 必须添加 “C” 关键字, 以阻止 C++ 的名字修饰:

```
extern "C" bool AsmFindArray( long n, long array[], long count );
```

名字修饰 (name decoration) 是一种标准的 C++ 编译器技术, 该技术通过在函数名字后面添加特殊字符来表示每个函数参数的确切类型, 任何支持函数重载 (两个函数名字相同, 但参数列表不同) 的语言都要求使用该技术。以汇编语言程序员的观点来看, 名字修饰意味着生成可执行文件时, C++ 编译器通知链接器查找并使用修饰过的名字而不是原始的未经修饰的名字。

12.3.1 使用汇编语言优化 C++ 代码

使用汇编语言优化其他语言编写的程序的方法之一是寻找速度瓶颈究竟在哪里。循环是典型的优化目标, 因为循环中的任何多余的语句都可能重复执行多次, 以至于对于程序性能产生显著的影响。

大多数 C/C++ 编译器都有生成 C/C++ 程序的汇编语言列表文件的命令行选项。例如在 Microsoft Visual C++ 中, 列表文件中可包含 C++ 源代码、汇编代码和机器码, 选项如表 12.1 所示, 其中最有用的选项或许是 /FAs 了, 它可用来显示 C++ 语句是如何翻译成汇编语言的。

表 12.1 Visual C++ 的汇编代码生成选项

命令行	列表文件内容
/FA	只有汇编代码
/FAc	汇编代码和机器码
/FAs	汇编代码和源代码
/FAcs	汇编代码、机器码和源代码

例子 FindArray

下面编写一个程序演示 C++ 编译器如何为一个名为 FindArray 的函数生成代码。接下来，我们试图编写一个比 C++ 编译器生成的代码更加高效的汇编语言版本的 FindArray。下面的 FindArray 函数 (C++) 在一个长整数类型的数组中查找一个值：

```
bool FindArray(long searchVal, long array[], long count)
{
    for (int i = 0; i < count; i++)
        if(array[i] == searchVal) return true;
    return false;
}
```

Visual C++ 为 FindArray 生成的代码

让我们看看 Visual C++ 为 FindArray 函数生成的汇编语言代码，其中同时包含 C++ 的源代码和汇编代码。该过程以发布 (Release) 模式编译，没有进行代码优化：

; Listing generated by Microsoft (R) Optimizing Compiler Version 14.00.50727.42

```
PUBLIC _FindArray
; Function compile flags: /Odtp

_TEXT SEGMENT
_i$2542 = -4 ; size = 4
_searchVal$ = 8 ; size = 4
_array$ = 12 ; size = 4
_count$ = 16 ; size = 4
_FindArray PROC

; 9 : {

    pushebp
    mov ebp, esp
    pushecx

; 10 : for(int i = 0; i < count; i++)

    mov DWORD PTR _i$2542[ebp], 0
    jmp SHORT $LN4@FindArray
$LN3@FindArray:
    mov eax, DWORD PTR _i$2542[ebp]
    add eax, 1
    mov DWORD PTR _i$2542[ebp], eax
$LN4@FindArray:
    mov ecx, DWORD PTR _i$2542[ebp]
    cmp ecx, DWORD PTR _count$[ebp]
    jge SHORT $LN2@FindArray

; 11 : {
; 12 :         if( array[i] == searchVal )

    mov edx, DWORD PTR _i$2542[ebp]
    mov eax, DWORD PTR _array$[ebp]
```

```

mov ecx, DWORD PTR [eax+edx*4]
cmp ecx, DWORD PTR _searchVal$[ebp]
jne SHORT $LN1@FindArray

; 13 :return true;

mov al, 1
jmp SHORT $LN5@FindArray
$LN1@FindArray:

; 14 :}

jmp SHORT $LN3@FindArray
$LN2@FindArray:

; 15 :
; 16 : return false;

xor al, al
$LN5@FindArray:

; 17 :}

mov esp, ebp
pop ebp
ret 0
_FindArray ENDP

```

三个 32 位的参数以下列顺序压入堆栈: count, array 和 searchVal, 三个参数中只有 array 是以传引用方式传递的, 因为在 C++ 中, 数组名就是指向数组第一个元素的隐式指针。过程在堆栈上保存 EBP 后, 压入了一个双字为局部变量 i 保留空间 (参见图 12.1)。

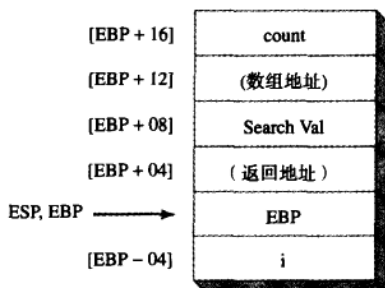


图 12.1 FindArray 函数的堆栈框架

在过程中, 汇编器通过压入 ECX 为变量 i 保留局部堆栈空间, 该存储空间在过程结束时 EBP 复制回 ESP 时释放。我们可以轻易地写出一个比该过程更加高效的汇编语言过程。

MASM 与 Visual C++ 相链接

下面编写一个人工优化的 FindArray 版本, 名为 AsmFindArray, 优化代码时遵循以下几条基本原则:

- 把尽可能多的处理置于循环之外。
- 把堆栈参数和局部变量放入寄存器中。
- 利用特殊的字符串/数组处理指令 (本例中是 SCASD)。

我们使用 Microsoft Visual C++ (Visual Studio) 编译 C++ 调用程序, 用 MASM 汇编被调用过程。Visual C++ 生成只能在保护模式下运行的 32 位应用程序, 本例的目标应用程序类型选择的是 Win32 控制台。在 Visual C++ 中, 8 位值在 AL 中返回, 16 位值在 AX 中返回, 32 位值在 EAX 中返回, 64 位值在 EDX:EAX 中返回。更大一点的数据结构 (结构值、数组等) 存储在静态数据区内, EAX 中返回指向该数据区的指针。

人工编写的汇编语言代码比 C++ 编译器生成的汇编代码更易读一些, 因为我们可以使用有意义的标号名称, 并定义常量以简化堆栈参数的使用。下面是完整的程序清单。

```
TITLE AsmFindArray Procedure      (AsmFindArray.asm)

.586
.model flat,C

AsmFindArray PROTO,
    srchVal:DWORD, arrayPtr:PTR DWORD, count:DWORD

.code
;-----
AsmFindArray PROC USES edi,
    srchVal:DWORD, arrayPtr:PTR DWORD, count:DWORD
;
; Performs a linear search for a 32-bit integer
; in an array of integers. Returns a boolean
; value in AL indicating if the integer was found.
;-----
    true = 1
    false = 0

    mov     eax,srchVal          ; 搜索值
    mov     ecx,count           ; 项目的数量
    mov     edi,arrayPtr        ; 数组的指针

    repne scasd                 ; 搜索
    jz      returnTrue          ; 如果找到 ZF = 1

returnFalse:
    mov     al,false
    jmp     short exit

returnTrue:
    mov     al,true

exit:
    ret
AsmFindArray ENDP
END
```

测试 FindArray 的性能

测试程序: 测试对比汇编语言编写的代码和以 C++ 编写的 (功能) 等价代码是非常有意思的事情。下面的 C++ 测试程序读入一个待查找的值并循环调用 FindArray 函数 100 万次, 在循环之前和循环之后都要获取系统时间。对 AsmFindArray 也要进行同样的测试。下面是 findarr.h 头文件, 其中包含了汇编语言过程和 C++ 函数的原型声明:

```
// findarr.h
extern "C" {
    bool AsmFindArray( long n, long array[], long count );
```

```
// 汇编语言版本
bool FindArray( long n, long array[], long count );
// C++版本
}
```

C++主模块：下面是调用 FindArray 和 AsmFindArray 的启动程序的清单 main.cpp:

```
// main.cpp - 测试 FindArray 和 AsmFindArray
#include <iostream>
#include <time.h>
#include "findarr.h"
using namespace std;

int main()
{
    // 以伪随机数填充数组
    const unsigned ARRAY_SIZE = 10000;
    const unsigned LOOP_SIZE = 1000000;

    long array[ARRAY_SIZE];
    for(unsigned i = 0; i < ARRAY_SIZE; i++)
        array[i] = rand();

    long searchVal;
    time_t startTime, endTime;
    cout << "Enter value to find: ";
    cin >> searchVal;
    cout << "Please wait. This will take between 10 and 30 seconds...\n";

    // 测试 C++ 函数
    time( &startTime );
    bool found = false;
    for( int n = 0; n < LOOP_SIZE; n++)
        found = FindArray( searchVal, array, ARRAY_SIZE );
    time( &endTime );
    cout << "Elapsed CPP time: " << long(endTime - startTime)
        << " seconds. Found = " << found << endl;

    // 测试汇编语言过程
    time( &startTime );
    found = false;
    for( int n = 0; n < LOOP_SIZE; n++)
        found = AsmFindArray( searchVal, array, ARRAY_SIZE );
    time( &endTime );
    cout << "Elapsed ASM time: " << long(endTime - startTime)
        << " seconds. Found = " << found << endl;

    return 0;
}
```

汇编语言代码和未优化的 C++ 代码的对比：以发布（Release）模式编译该 C++ 程序，关闭代码优化选项。下面是在最坏的情况下（值未找到）程序的输出：

```
Enter value to find: 55
Elapsed CPP time: 28 seconds. Found = 0
Elapsed ASM time: 14 seconds. Found = 0
```

汇编语言代码和优化的 C++ 代码的对比：接下来，我们设置编译器选项，针对程序代码的执行速度进行优化，然后再次运行测试程序。下面是程序输出的结果，这表明人工编写的汇编代码

比编译器优化过的 C++ 代码还是明显地快一些^①：

```
Enter value to find: 55
Elapsed CPP time: 20 seconds. Found = 0
Elapsed ASM time: 14 seconds. Found = 0
```

指针和下标

使用旧版本 C 编译器的程序员注意到使用指针处理数组要比使用下标更加高效。例如，下面这个版本的 FindArray 就使用了指针：

```
bool FindArray( long searchVal, long array[], long count )
{
    long * p = array;
    for(i = 0; i < count; i++, p++)
        if( searchVal == *p )
            return true;
    return false;
}
```

Visual C++ 编译器为使用指针的 FindArray 版本生成的代码和为前面使用下标的版本生成的汇编代码几乎是相同的，这是由于现代的编译器已经很擅长做代码优化了，使用指针不再比使用下标更加高效。下面是 C++ 编译器为 FindArray 生成的代码中的循环部分：

```
$L176:
    cmp     esi, DWORD PTR [ecx]
    je      SHORT $L184
    inc     eax
    add     ecx, 4
    cmp     eax, edx
    jnl     SHORT $L176
```

高级语言编译器在代码优化方面所做的工作是卓有成效的，读者应该多花时间研究 C++ 编译器的输出，从中学习代码优化、参数传递和目标代码实现等技术。事实上，很多计算机科学系的学生都要学习包含这些主题的编译器构建课程。另外还有一项很重要的一点是，读者应该意识到编译器通常采用常规的优化处理手段，因为编译器对个别的特殊应用和安装的特殊硬件并不了解。也有一些编译器针对特定的处理器（如奔腾处理器）进行了优化，可极大地提高编译后程序的执行速度。手动编写的汇编语言代码可以充分利用计算机系统的某些硬件的特性，比如视频显示卡、声卡和数据采集卡等。

12.3.2 调用 C 和 C++ 函数

读者还可以编写调用 C++ 函数的汇编语言代码。这样做的原因至少包括：

- C++ 中输入和输出更加灵活，这是由于 C++ 有内容异常丰富的输入输出流（iostream）库，这类库在处理浮点数时特别有用。
- C++ 中有扩展的数学运算函数库。

在调用标准 C/C++ 库时，必须从 C/C++ 的主模块启动程序以运行库的初始化代码。

函数原型

汇编语言中调用的 C++ 函数定义中必须包含“C”和 extern 关键字。下面是基本格式：

```
extern "C" funcName( paramlist )
{ . . . }
```

^① 根据原书的勘误表，在 VS 2005 编译器下，CPP 时间应为 11 s，因此这个结论不再成立——编者注。

下面是一个例子：

```
extern "C" int askForInteger( )
{
    cout << "Please enter an integer:";
    //...
}
```

与其修改每个函数的定义，不如把一组函数的声明放在 extern “C” 语句块中，这样在定义函数时就可以省略掉 extern 和 “C” 关键字了：

```
extern "C" {
    int askForInteger();
    int showInt( int value, unsigned outWidth );
    etc.
}
```

汇编语言模块

使用 Irvine32 链接库：如果调用的汇编语言模块使用了 Irvine32 库中的过程，一定要清楚该库使用了如下的.MODEL 伪指令：

```
.model flat, STDCALL
```

虽然 STDCALL 调用约定与 Win32 API 兼容，但与 C 程序使用的调用约定并不匹配。因此在声明汇编语言模块调用的外部的 C/C++ 函数时，必须在 PROTO 伪指令后添加一个 C 修饰符：

```
INCLUDE Irvine32.inc
askForInteger PROTO C
showInt PROTO C, value:SDWORD, outWidth:DWORD,
```

由于链接器必须把汇编语言中声明的函数名/参数列表同 C++ 模块导出的函数进行匹配，C 修饰符是必需的。除此之外，汇编器必须在调用完使用 C 调用约定（参见 8.4.1 节）的函数后正确地清理堆栈，这也是原因之一。

汇编语言模块中要有 C++ 调用的汇编语言过程，同样也必须使用 C 修饰符，以使汇编器使用链接器能够识别的命名约定。例如，下面的 SetTextColor 过程只包含一个双字参数，就使用了 C 修饰符：

```
SetTextOutColor PROC C,
    color:DWORD
.
SetTextOutColor ENDP
```

最后，如果你编写的汇编语言代码调用了其他使用 C 调用约定的汇编语言过程，那么在过程调用完毕后必须负责清除堆栈上的参数。

使用.MODEL 伪指令：如果汇编语言模块没有调用 Irvine32 库中的过程，那么可以使用.MODEL 伪指令指定使用 C 调用约定：

```
; (do not INCLUDE Irvine32.inc)
.586
.model flat, C
```

现在对于 PROTO 和 PROC 伪指令来说，就不必再添加 C 修饰符了：

```
askForInteger PROTO
showInt PROTO, value:SDWORD, outWidth:DWORD

SetTextOutColor PROC,
    color:DWORD
.
SetTextOutColor ENDP
```

函数的返回值

C++语言规范并没有规定实现细节,因此 C++返回值的方式没有标准。在编写调用 C++函数的汇编代码时,应该查阅 C++编译器的文档,弄清 C++函数究竟是如何返回值的。下面是 C++函数几种可能的返回值的方式,注意不是全部。

- 整数可在单个寄存器或一组寄存器中返回。
- 可以由调用者在堆栈上为返回值保留空间,被调用函数在返回前把返回值插入保留的堆栈空间。
- 在函数返回前要返回的浮点值通常压入浮点栈。

下面列出了 Microsoft Visual C++是如何返回值的。

- bool 和 char 值在 AL 中返回。
- short int 在 AX 中返回。
- int 和 long int 在 EAX 中返回。
- 指针在 EAX 中返回。
- float, double 和 long double 分别按 4, 8 和 10 字节压入浮点栈。

12.3.3 乘法表的例子

下面编写一个程序提示用户输入一个整数,然后通过移位对该整数分别乘以递增的 2 的次幂 ($2^1 \sim 2^{10}$),然后显示乘积,乘积开头以空格填充对齐。我们使用 C++函数进行输入输出,汇编语言模块中调用了两个 C++函数,整个程序的主模块是用 C++编写的。

汇编语言模块

汇编语言模块中包含了一个名为 DisplayTable 的过程,该过程调用了名为 askForInteger 的 C++函数从用户那里读入一个整数,过程在一个循环中重复左移整数 intVal 并调用 showInt 显示该整数。

```
; ASM function called from C++
INCLUDE Irvine32.inc

; External C++ functions:
askForInteger PROTO C
showInt PROTO C, value:SDWORD, outWidth:DWORD
newLine PROTO C

OUT_WIDTH = 8
ENDING_POWER = 10

.data
intVal DWORD ?

.code
;-----
SetTextOutColor PROC C,
    color:DWORD
;
; Sets the text colors and clears the console
; window. Calls Irvine32 library functions.
;-----
    mov     eax,color
    call    SetTextColor
    call    clrscr
    ret
```

```

SetTextOutColor ENDP
;-----
DisplayTable PROC C
;
; Inputs an integer n and displays a
; multiplication table ranging from n * 2^1
; to n * 2^10.
;-----
    INVOKE askForInteger          ; 调用 C++ 函数
    mov     intVal, eax           ; 保存整数
    mov     ecx, ENDING_POWER    ; 循环计数器

L1:  push    ecx                  ; 保存循环计数器
     shl     intVal, 1           ; 乘以 2
     INVOKE showInt, intVal, OUT_WIDTH
     INVOKE newLine              ; 输出 CR/LF
     pop     ecx                  ; 恢复循环计数器
     loop   L1

    ret
DisplayTable ENDP
END

```

在 DisplayTable 过程中，必须在调用 showInt 和 newLine 函数的前后保存/恢复 ECX 的值，这是由于 Visual C++ 的函数并不会保存/恢复通用寄存器。askForInteger 函数在 EAX 中返回结果。

在调用 C++ 函数时并不一定要求使用 INVOKE，使用 PUSH 和 CALL 指令也能达到同样的效果。下面演示了如何不用 INVOKE 调用函数 showInt：

```

push    OUT_WIDTH                ; 最后一个参数先压栈
push    intVal
call    showInt                  ; 调用函数
add     esp, 8                   ; 清理堆栈

```

此时必须遵守 C 调用约定：参数按照与参数列表相反的顺序（从右到左）压栈，调用者在调用后负责从堆栈上删除参数。

C++ 启动程序

下面来看一下启动程序的 C++ 模块，其入口点是 main()，在入口点时已经确保执行了必需的 C++ 初始化代码了。启动模块中包含了外部汇编语言过程和三个导出函数的函数原型：

```

// main.cpp
// 解释 C++ 程序和外部汇编语言模块之间
// 的函数调用

#include <iostream>
#include <iomanip>
using namespace std;

extern "C" {
    // 外部的汇编语言过程
    void DisplayTable();
    void SetTextOutColor(unsigned color);

    // 局部 C++ 函数
    int askForInteger();
    void showInt(int value, int width);
}

// 程序入口点
int main()
{

```

```

    SetTextOutColor( 0x1E );           // 蓝底黄字
    DisplayTable();                   // 调用汇编语言过程
    return 0;
}

// 提示用户输入一个整数
int askForInteger()
{
    int n;
    cout << "Enter an integer between 1 and 90,000:";
    cin >> n;
    return n;
}

// 以指定的宽度显示一个有符号整数
void showInt( int value, int width )
{
    cout << setw(width) << value;
}

```

构建项目: 在本书的 Web 站点上 (www.asmirvine.com) 有一份构建 C++/汇编语言项目的教程。

程序的输出: 下面是在用户输入 90 000 时, 乘法表程序的输出示例:

```

Enter an integer between 1 and 90,000: 90000
180000
360000
720000
1440000
2880000
5760000
11520000
23040000
46080000
92160000

```

Visual Studio 的项目属性

如果读者使用 Visual Studio 构建包含 C++ 并且调用了 Irvine32 库的汇编语言模块的程序, 则需要修改一些项目属性。我们以 Multiplication_Table 程序为例进行说明: 选择 Project 菜单下的 Properties 子菜单, 在对话框的 Configuration 下拉列表中选择 All Configurations, 在 Configurations Properties 下双击 Linker 并选择 Command Line, 在 Additional Options 中添加 C:\Irvine32\Irvine32.lib (见图 12.2)。点击 OK 关闭属性页窗口, 现在 Visual Studio 就能够找到 Irvine32 库了。

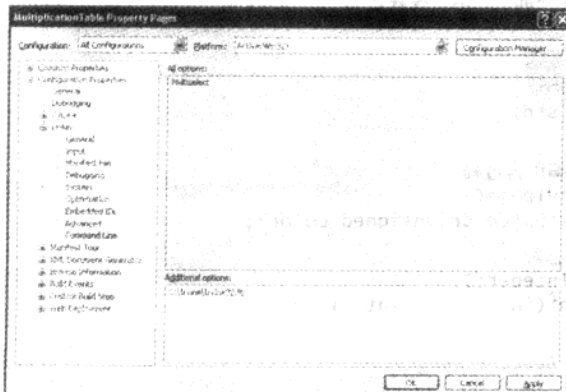


图 12.2 链接器的命令行属性页

这里给出的信息是在 Visual Studio 2005 下测试的。随着版本的变化,信息可能有所不同,请查阅本书的 Web 站点上(www.asmirvine.com)的更新。

12.3.4 调用 C 库函数

C 语言有一系列的标准函数,这些函数的集合称为标准 C 库(Standard C Library)。标准 C 库中函数对 C++ 程序也是可用的,对于附加到 C/C++ 程序中的汇编语言模块而言,这些函数同样可用。汇编语言模块必须包含其调用的每个 C 函数的原型声明。通常可以在 C++ 编译器手册中找到相应的 C 函数原型定义,在汇编语言模块中调用相应的 C 函数之前,必须把它的函数原型翻译成汇编语言格式的原型声明。

printf 函数:下面是 printf 函数的 C/C++ 函数原型,它的第一个参数是一个字符指针,后面跟可变数目的参数:

```
int printf(  
    const char *format [, argument]...  
);
```

(C++ 编译器的帮助中有 printf 函数的详尽的说明文档。)汇编语言中等价的原型声明把 char * 改成 PTR BYTE,把可变长度参数列表改成 VARARG 类型:

```
printf PROTO C, pString:PTR BYTE, args:VARARG
```

另外一个有用的函数是 scanf,该函数可以从标准输入(键盘)读入一个字符、数字或字符串并把输入赋值给变量:

```
scanf PROTO C, format:PTR BYTE, args:VARARG
```

用 printf 函数显示格式化实数

用汇编语言编写格式化并显示浮点数的函数并不轻松,不过可以利用 C/C++ 语言中的 printf 库函数做这种工作,而非自己亲自完成。读者必须首先创建一个 C/C++ 启动模块,然后再把这个模块同汇编语言代码链接起来。下面是如何在 Visual C++ 中设置该程序框架的步骤。

1. 在 Visual C++ 中创建一个 Win32 控制台程序,创建一个 main.cpp 的文件,在文件中编写一个 main 函数,调用 asmMain。在 main 函数中要声明至少一个浮点数变量。在下面的代码中,声明 d 的语句除了强制 Visual C++ 加载其浮点运行时库之外并没有别的作用^①。

```
extern "C" void asmMain();  
  
int main()  
{  
    double d = 3.5;           // 加载浮点库  
    asmMain();  
    return 0;  
}
```

2. 在 main.cpp 的同一目录下,创建一个汇编语言模块 asmMain.asm,该模块包含一个名为 asmMain 的过程,声明使用 C 调用约定:

```
TITLE asmMain.asm  
.386  
.model flat,stdcall  
.stack 2000  
.code  
asmMain PROC C  
  
    ret  
asmMain ENDP  
END
```

^① 在我们测试的 Visual C++ 版本中,必须包含至少一条使用浮点值的语句以强制 Visual C++ 加载浮点模块,否则就会发生错误“floating point not loaded”。

3. 汇编 `asmMain.asm` (不链接), 生成 `asmMain.obj`。
 4. 把 `asmMain.obj` 添加到 C++ 项目中。
 5. 构建并运行项目, 如果修改了 `asmMain.asm`, 则重新汇编该模块并重新构建项目。
- 一旦程序框架正确设置之后, 就可以向 `asmMain.asm` 中添加调用 C/C++ 语言函数的代码了。

显示双精度值: 下面的 `asmMain` 中的汇编语言代码通过调用 `printf` 打印出一个 `REAL8` 类型的实数值:

```
.data
double1 REAL8 1234567.890123
formatStr BYTE "%.3f",0dh,0ah,0
.code
INVOKE printf, ADDR formatStr, double1
```

下面是对应的输出:

```
1234567.890
```

传递给 `printf` 的格式化串与 C++ 中稍有不同, 不能在字符串中使用特殊的转义字符 (如 `\n`), 必须直接插入其对应的 ASCII 值 (`0dh,0ah`)。

传递给 `printf` 的浮点参数应该声明为 `REAL8`, 虽然传递 `REAL4` 类型的值是可能的, 但这需要编写很多额外的代码。读者可以查看 C++ 编译器是如何做的: 声明一个 `float` 类型的变量并把它传递给 `printf`, 然后编译该程序并使用调试器跟踪程序的反汇编代码。

多个参数: `printf` 函数接收可变数目的参数, 因此可以在一次函数调用中格式化并显示两个数字, 这非常容易:

```
TAB = 9
.data
formatTwo BYTE "%.2f",TAB,"%.3f",0dh,0ah,0
val1 REAL8 456.789
val2 REAL8 864.231
.code
INVOKE printf, ADDR formatTwo, val1, val2
```

下面是对应的输出:

```
456.79 864.231
```

(参见本书附带代码中 `Examples\ch12\VisualCPP` 目录下的项目 `Printf_Example`。)

用 `scanf` 函数输入实数

可以调用 `scanf` 函数从用户那里读入一个浮点值。下面的函数原型在 `SmallWin.inc` 中定义 (已由 `Irvine32.inc` 包含了):

```
scanf PROTO C,
    format:PTR BYTE, args:VARARG
```

应该向该函数传递一个格式化串的偏移地址, 以及一个或多个 `REAL4` 或 `REAL8` 类型的变量的偏移地址, 以存放用户输入的值。调用示例如下:

```
.data
strSingle BYTE "%f",0
strDouble BYTE "%lf",0
single1 REAL4 ?
```

```
double1 REAL8 ?
.code
INVOKE scanf, ADDR strSingle, ADDR single1
INVOKE scanf, ADDR strDouble, ADDR double1
```

必须从 C/C++ 启动程序中调用该汇编语言代码。

12.3.5 目录列表程序

下面再来编写一个小程序，清除屏幕，显示当前目录并要求用户输入一个文件名（读者可能想扩展该程序，以便能够打开并显示选定的文件）。

C++代理模块：C++模块值包含了对 `asm_main` 的一次调用，因此我们称为一个代理模块：

```
// main.cpp
// stub module: launches assembly language program
extern "C" void asm_main();           // asm startup proc

void main()
{
    asm_main();
}
```

汇编语言模块：汇编语言模块包含了函数的原型声明、几个字符串以及一个变量 `fileName`，它两次调用了 `system` 函数，向 `system` 函数传递 `cls` 和 `dir` 命令，然后调用 `printf` 显示一条提示要求用户输入文件名，接下来调用 `scanf` 函数读入用户输入的文件名。汇编语言模块没有调用 Irvine32 库中的过程，因此可以在 `.MODEL` 伪指令中使用 C 语言调用约定：

```
; ASM program launched from C++      (asmMain.asm)
.586
.MODEL flat,C
; Standard C library functions:
system PROTO, pCommand:PTR BYTE
printf PROTO, pString:PTR BYTE, args:VARARG
scanf PROTO, pFormat:PTR BYTE, pBuffer:PTR BYTE, args:VARARG
fopen PROTO, mode:PTR BYTE, filename:PTR BYTE
fclose PROTO, pFile:DWORD

BUFFER_SIZE = 5000
.data
str1 BYTE "cls",0
str2 BYTE "dir/w",0
str3 BYTE "Enter the name of a file:",0
str4 BYTE "%s",0
str5 BYTE "cannot open file",0dh,0ah,0
str6 BYTE "The file has been opened",0dh,0ah,0
modeStr BYTE "r",0

fileName BYTE 60 DUP(0)
pBuf DWORD ?
pFile DWORD ?

.code
asm_main PROC

    ; clear the screen, display disk directory
    INVOKE system, ADDR str1
    INVOKE system, ADDR str2

    ; ask for a filename
    INVOKE printf, ADDR str3
    INVOKE scanf, ADDR str4, ADDR fileName
```

```
; try to open the file
INVOKE fopen, ADDR fileName, ADDR modeStr
mov    pFile,eax

; IF eax == 0 ; cannot open file?
INVOKE printf,ADDR str5
jmp quit
.ELSE
INVOKE printf,ADDR str6
.ENDIF

; Close the file
INVOKE fclose, pFile

quit:
    ret ; return to C++ main
asm_main ENDP
END
```

scanf 要求输入两个参数：第一个是一个指向格式化字符串的指针 (“%s”), 第二个参数是一个指向输入字符变量 (fileName) 的指针。关于 C 的标准函数, 由于在 Web 上有非常多的文档, 因此这里就不再花时间解释了。一份非常好的参考资料是 Brian W. Kernighan 和 Dennis M. Ritchie 编写的 *The C Programming Language*, 第二版, 它由 Prentice Hall 出版社于 1988 年出版。

12.3.6 本节习题

1. 如果 C++ 中的函数要被外部的汇编语言模块调用, 那么函数定义中必须包含哪两个 C++ 关键字?
2. Irvine32 库使用的调用约定与 C/C++ 语言使用的调用约定有哪些地方是不兼容的?
3. C++ 函数通常如何返回浮点值?
4. 在 Microsoft Visual C++ 中, 函数如何返回 short int?
5. 标准 C 函数 printf 对应的有效的汇编语言原型声明是什么?
6. 下面的 C 语言函数被调用时, 参数 x 是最先还是最后压入堆栈?

```
void MySub( x, y, z );
```

7. 在 C++ 中, 要调用的外部过程的 extern 声明中 “C” 关键字有什么作用?
8. 在 C++ 中调用外部汇编语言过程时, 为什么注意名字修饰的问题是非常重要的?
9. 本章中, 对于用数组下标的方式编写的循环和以指针变量的方式编写的循环, 一个能够进行优化的 C++ 编译器为这二者生成的代码有什么不同?

12.4 在实地址模式下与 C/C++ 程序链接

许多嵌入式系统应用程序仍然是为使用 Intel 8086/8088 的 16 位环境编写的, 此外, 还有一些使用 32 位处理器的应用运行于实地址模式下。因此给出一些在实地址模式环境下在 C/C++ 中调用汇编语言过程的例子就非常重要了。

本节中的例子使用 16 位版本的 Borland C++ 5.01, 同时选用 Windows 98 (MS-DOS 窗口) 作为目标操作系统并采用小型内存模式。我们使用 Borland TASM 4.0 编译这些例子, 因为大部分 Borland C++ 用户更愿意使用 TASM 而不是 MASM。本章中我们使用 Borland C++ 创建 16 位实模式程序, 解释小型内存模式和大型内存模式这两种模式, 并讲述如何调用近过程和远过程。

12.4.1 与 Borland C++程序链接

函数的返回值：在 Borland C++中，函数使用 AX 返回 16 位值，使用 DX:AX 返回 32 位值。更大的数据结构（如结构和数组等）被存储在静态数据区中，并在 AX 中返回指向该数据区的指针（在中型、大型和巨型模式中，使用 DX:AX 返回 32 位的指针值）。

创建一个项目：读者可以在 Borland C++集成开发环境中（IDE）创建一个新项目，然后创建一个源代码模块文件（CPP 文件）并输入 C++主程序的代码。接下来创建包含要调用的汇编语言过程的汇编源代码文件，读者可以在 MS-DOS 命令行下（或者通过 Borland C++ IDE 中的转换功能）使用 TASM 把汇编语言源文件编译为目标文件。源代码文件的名字（不包括扩展名）不能多于 8 个字符，否则 16 位的链接器无法识别。

如果已经单独编译了汇编语言模块，那么把汇编器生成的目标文件添加到 C++项目中，然后调用菜单中的 MAKE 或 BUILD 命令，这些命令将编译 CPP 文件。如果没有错误，链接器就把两个目标模块链接起来生成一个可执行程序。建议：CPP 文件名长度不要超过 8 个字符，否则使用 DOS 下的 Turbo Debugger 调试程序的时候会找不到文件名。

调试：Borland C++编译器不允许从 IDE 中运行 DOS 调试器，读者应该在 DOS 命令提示行下或从 Windows 桌面上运行 Turbo Debugger 调试器，在调试器中通过 File/Open 菜单项选择 C++链接器创建的可执行文件，这时 C++源代码应当立即显示出来，进而可以运行并跟踪程序了。

保护寄存器：由 Borland C++程序调用的汇编语言过程必须保护 BP, DS, SS, SI, DI 和方向标志的值。

存储尺寸：16 位 Borland C++程序的所有数据类型都有特定的存储尺寸，这些类型的尺寸与 Borland C++的实现有关，对于其他不同的 C++编译器需要做相应的调整。请参考表 12.2。

表 12.2 16 位程序中的 Borland C++数据类型

C++类型	存储字节数	汇编语言类型
char, unsigned char	1	byte
int, unsigned int, short int	2	word
enum	2	word
long, unsigned long	4	dword
float	4	dword
double	8	qword
long double	10	tbyte
near pointer	2	word
far pointer	4	dword

12.4.2 例子：ReadSector

（该例子必须在 MS-DOS 或 Windows 98/98/Me 下运行。）下面是一个调用外部汇编语言过程 ReadSector 的 Borland C++程序。C++编译器通常并不包含读磁盘扇区的库函数，因为这一类细节的实现和硬件过于相关，为所有的计算机都实现这种库是不切实际的。汇编语言程序通过调用 INT 21h 的 7305h 功能（细节参见 14.4 节）可轻易地读取磁盘扇区，当前的任务就是创建汇编语言与 C++之间的接口，以结合两种语言的优势。

例子 ReadSector 要求使用 16 位编译器, 因为程序中包含了 MS-DOS 中断调用 (从 32 位程序中调用 16 位中断也是可能的, 但是这超出了本书的范围^①)。最后一个能生成 16 位程序的 Visual C++ 的版本是 1.5, 其他能生成 16 位代码的编译器还有 Turbo C 和 Turbo Pascal, 两者都是 Borland 的产品。

程序的执行: 首先解释一下程序的执行。当程序启动的时候, 用户可以选择要读取的驱动器号、起始扇区和扇区数, 例如下面这位用户想读取 A 驱动器的 0~19 扇区:

```
Sector display program.
Enter drive number [1=A, 2=B, 3=C, 4=D, 5=E,...]: 1
Starting sector number to read: 0
Number of sectors to read: 20
```

这些信息被传递给汇编语言过程, 汇编语言过程把数据读入缓冲区, 然后 C++ 程序开始以每次显示一个缓冲区的方式来显示缓冲区的内容。非 ASCII 字符在显示时以点代替。例如, 下面是程序显示的 A 驱动器扇区 0 的内容:

```
Reading sectors 0 - 20 from Drive 1
Sector 0 -----
.<.(P3j2IHC.....@.....)Y...MYDISK   FAT12   .3.
...{...x..v..V..U..".~..N.....|.E...F..E.8N$}"...w.r...:f..
|f;..W.u....V....s.3..F...f..F..V..F...v..`F..V.. ...^...H...F
..N.a....#..r98-t.`....}.at9Nt... ;.r....}......t.<.t.....
..}....}.^..f.....}.E..N...F..V.....r...p..B.-`fj.RP.Sj
.j...t...3..v...v.B...V.....VS...d.ar.@u.B.^Iuw....'..I
nvalid system disk...Disk I/O error...Replace the disk, and then
press any key...IOSYMSDOS  SYS...A....~...@...U.
```

所有扇区的内容一个接一个地顺序显示, 直到整个缓冲区显示完毕为止。

调用 ReadSector 的 C++ 主程序

下面是调用 ReadSector 过程的完整的 C++ 程序:

```
// main.cpp - Calls the ReadSector Procedure
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
const int SECTOR_SIZE = 512;

extern "C" ReadSector( char * buffer, long startSector,
                      int driveNum, int numSectors );

void DisplayBuffer( const char * buffer, long startSector,
                   int numSectors )
{
    int n = 0;
    long last = startSector + numSectors;
    for(long sNum = startSector; sNum < last; sNum++)
    {
        cout << "\nSector " << sNum
              << " -----"
              << "-----\n";
        for(int i = 0; i < SECTOR_SIZE; i++)
        {
            char ch = buffer[n++];
```

① 参见 Barry Kauler 的 *Windows Assembly Language and System Programming*, CMP books, 1997。

```

        if( unsigned(ch) < 32 || unsigned(ch) > 127)
            cout << '.';
        else
            cout << ch;
    }
    cout << endl;
    getch();        // pause - wait for keypress
}
}

int main()
{
    char * buffer;
    long startSector;
    int driveNum;
    int numSectors;

    system("CLS");
    cout << "Sector display program.\n\n"
         << "Enter drive n: 暂停 - 等待用户按键";
    cin >> driveNum;
    cout << "Starting sector number to read: ";
    cin >> startSector;
    cout << "Number of sectors to read:";
    cin >> numSectors;
    buffer = new char[numSectors * SECTOR_SIZE];

    cout << "\n\nReading sectors" << startSector << " - "
         << (startSector + numSectors) << "from Drive"
         << driveNum << endl;

    ReadSector( buffer, startSector, driveNum, numSectors );
    DisplayBuffer( buffer, startSector, numSectors );
    system("CLS");
    return 0;
}

```

在程序的顶部可以找到 ReadSector 函数的声明，或者称为原型：

```
extern "C" ReadSector( char buffer[], long startSector,
                      int driveNum, int numSectors );
```

第一个参数 buffer 指向存放从磁盘读取的数据的缓冲区，第二个参数 startSector 是要读的起始扇区号，第三个参数 driveNum 是磁盘的驱动器号，第四个参数 numSectors 是要读取的扇区数目。其中第一个参数是以传引用方式传递的，其他参数都是以传值方式传递的。

程序在 main 中提示用户输入驱动器号、起始扇区和要读取的扇区数目，并动态分配存放扇区数据的缓冲区：

```

cout << "Sector display program.\n\n"
     << "Enter drive number [1=A, 2=B, 3=C, 4=D, 5=E,...]: ";
cin >> driveNum;
cout << "Starting sector number to read:";
cin >> startSector;
cout << "Number of sectors to read:";
cin >> numSectors;
buffer = new char[numSectors * SECTOR_SIZE];

```

这些信息被传递给外部的 ReadSector 过程，该过程从磁盘读取数据并填充缓冲区：

```
ReadSector( buffer, startSector, driveNum, numSectors );
```

然后缓冲区被传递给了 DisplayBuffer，这个 C++ 函数以 ASCII 文本方式显示每个扇区的数据：

```
DisplayBuffer( buffer, startSector, numSectors );
```

汇编语言模块

包含 ReadSector 过程的汇编语言模块如下所示, 要注意的是: 由于程序是一个实模式程序, 因此 .386 伪指令必须出现在 .MODEL 伪指令之后, 以便让编译器生成 16 位的段:

```
TITLE Reading Disk Sectors                (ReadSec.asm)

; The ReadSector procedure is called from a 16-bit
; Real-mode application written in Borland C++ 5.01.
; It can read FAT12, FAT16, and FAT32 disks under
; MS-DOS, Windows 95, Windows 98, and Windows Me.
Public _ReadSector
.model small
.386

DiskIO STRUC
    strtSector DD ?           ; 起始扇区号
    nmSectors  DW 1           ; 扇区数量
    bufferOfs  DW ?           ; 缓冲区偏移地址
    bufferSeg   DW ?          ; 缓冲区所在的段
DiskIO ENDS

.data
diskStruct DiskIO <>

.code
;-----
_ReadSector PROC NEAR C
    ARG bufferPtr:WORD, startSector:DWORD, driveNumber:WORD, \
        numSectors:WORD
;
; Read n sectors from a specified disk drive.
; Receives: pointer to buffer that will hold the sector,
;   data, starting sector number, drive number,
;   and number of sectors.
; Returns: nothing
;-----
    enter 0,0
    pusha
    mov  eax,startSector
    mov  diskStruct.strtSector,eax
    mov  ax,numSectors
    mov  diskStruct.nmSectors,ax
    mov  ax,bufferPtr
    mov  diskStruct.bufferOfs,ax
    push ds
    pop  diskStruct.bufferSeg
    mov  ax,7305h           ; 绝对扇区读写
    mov  cx,0FFFFh         ; 必须是 0FFFFh
    mov  dx,driveNumber     ; 驱动器号
    mov  bx,OFFSET diskStruct ; 扇区号
    mov  si,0               ; 读取模式
    int  21h               ; 读取磁盘扇区
    popa
    leave
```

```
ret
_ReadSector ENDP
END
```

由于这个例子要用 Borland TASM 来编译, 因此使用了 Borland 的 ARG 关键字来指定过程参数, 注意 ARG 关键字允许以和 C++ 函数参数一致的顺序指定参数:

```
ASM:      _ReadSector PROC near C
          ARG bufferPtr:word, startSector:dword, \
            driveNumber:word, numSectors:word
C++:      extern "C" ReadSector(char buffer[],
          long startSector, int driveNum,
          int numSectors);
```

参数使用标准的 C 调用约定按反向顺序压入堆栈, 离 EBP 最近的 numSectors 参数是被压入堆栈的第一个参数, 堆栈框架如图 12.3 所示。StartSector 是一个 32 位的双字, 占用了堆栈上从 [BP + 6] 到 [BP + 9] 的位置。由于该程序是以小型内存模式编译的, 因此 buffer 以 16 位近指针的形式传递。

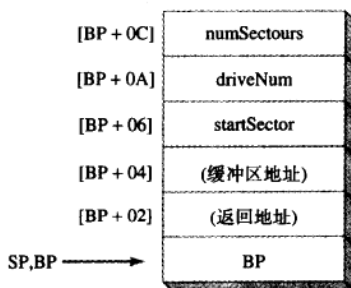


图 12.3 ReadSector 过程的堆栈框架

12.4.3 例子: 大随机整数

这里再给出一个从 Borland C++ 中调用外部函数的有用例子: 调用 LongRand 汇编语言函数, 该函数返回一个 32 位的伪随机无符号整数。这个函数非常有用, 因为 Borland C++ 中的 rand() 库函数返回的随机数是 0~RAND_MAX (32767) 之间的, 而 LongRand 过程可以返回 0~4 294 967 295 之间的一个整数。

程序使用大型内存模式编译, 该模式允许数据段超过 64 KB, 这就要求数据指针和返回值使用 32 位值。C++ 中的外部函数声明如下:

```
extern "C" unsigned long LongRandom();
```

下面是主程序的清单, 程序为数组 rArray 分配存储空间, 然后在循环中调用 LongRandom 函数并在数组中插入生成的随机数, 在标准输出上显示该随机数:

```
// main.cpp
// Calls the external LongRandom function, written in
// assembly language, that returns an unsigned 32-bit
// random integer. Compile in the Large memory model.

#include <iostream.h>
extern "C" unsigned long LongRandom();
const int ARRAY_SIZE = 500;

int main()
{
```



```

// Allocate array storage, fill with 32-bit
// unsigned random integers, and display:
unsigned long * rArray = new unsigned long[ARRAY_SIZE];
for(unsigned i = 0; i < ARRAY_SIZE; i++)
{
    rArray[i] = LongRandom();
    cout << rArray[i] << ' ';
}
cout << endl;
return 0;
}

```

LongRandom 函数: 包含 LongRandom 的汇编语言模块是由本书链接库中的 Random32 过程改写而成的:

```

; LongRandom procedure module                (longrand.asm)
.model large
.386
Public _LongRandom
.data
seed  DWORD 12345678h

; Return an unsigned pseudo-random 32-bit integer
; in DX:AX, in the range 0 - FFFFFFFFh.
.code
_LongRandom  PROC far, C
    mov     eax, 343FDh
    mul     seed
    xor     edx, edx
    add     eax, 269EC3h
    mov     seed, eax                ; 保存下一次调用的随机种子
    ror     eax, 8                    ; 把最低位的数字循环移出
    shld    edx, eax, 16              ; 复制 EAX 的高 16 位至 DX
    ret
_LongRandom  ENDP
end

```

ROR 指令有助于避免生成小随机数时发生重复。Borland C++ 要求 32 位的返回值存放在 DX:AX 中, 因此程序使用 SHLD 指令把 EAX 的高 16 位复制到 DX 中, SHLD 看起来似乎就是为方便地完成此类任务而设计的。

12.4.4 本节习题

1. Borland C++ 调用的汇编语言过程必须保护哪些寄存器和标志?
2. 在 Borland C++ 中, 下面的类型分别使用多少字节?
(1) int (2) enum (3) float (4) double
3. 在 ReadSector 过程中, 如果不使用 ARG 伪指令, 下面的语句应该如何编写?
mov eax, startSector
4. 如果把本节的 LongRandom 函数中的 ROR 指令去掉, 输出会发生什么变化?

12.5 本章小结

汇编语言是优化以高级语言编写的大型应用程序特定部分的理想工具, 也是为特定硬件定制过程的优秀工具。这些技术要用到下面两种方法:

- 在高级语言代码中编写内联的汇编语句。
- 把汇编语言过程同高级语言代码相链接。

两种方法各有优缺点，本章对两种方法都做了介绍。

一种语言使用的命名约定是指段和模块命名的方式，以及变量和过程命名的规则或特征。程序使用的内存模式决定了调用和引用是近的（在同一个段内）还是远的（在不同段内）。

在从其他语言编写的程序中调用汇编语言过程的时候，两种语言之间共享的任何标识符必须是兼容的，被调用过程必须使用同调用过程兼容的段名。过程的编写者使用高级语言调用约定来决定如何接收参数，调用约定还影响堆栈指针的恢复方式，即由调用过程恢复还是由被调用过程恢复。

在 Visual C++ 中，`__asm` 伪指令用于在 C++ 源程序中编写内联汇编代码，本章以一个文件加密程序说明了内联汇编语句的用法。

本章还讲述了如何将汇编语言过程同保护模式下的 Microsoft Visual C++ 程序及实地址模式下的 Borland C++ 程序相链接。

调用标准 C/C++ 库中的函数时，首先创建一个包含 main 函数的 C/C++ 代理程序，当运行 main 函数时，运行时库已经初始化好了。在 main 中可以调用汇编语言模块中的启动过程，汇编语言模块可以调用任何 C 标准库中的函数。

FindArray 过程是用汇编语言编写的，它被一个 Visual C++ 程序调用。为了学习更多的代码优化技巧，我们还比较了编译器生成的汇编源文件和手工编写的汇编代码之间的差别。ReadSector 程序展示了实地址模式的 Borland C++ 程序是如何调用汇编语言过程读取磁盘扇区的。

12.6 编程练习

1. 例子 MultArray

参考 12.3.1 节的例子 FindArray，编写一个汇编语言过程 MultArray，把一个双字数组的每个元素都乘以一个整数。以 C++ 编写一个同样功能的函数，然后编写一个测试程序，循环调用两个版本的 MultArray 并比较它们的执行时间。

2. ReadSector，十六进制显示

（要求使用一个能够在 MS-DOS 或 Windows 98/98/Me 下运行的 16 位实地址模式的 C++ 编译器。）为 12.4.2 节的 C++ 程序增加一个新的过程，该过程调用 ReadSector 过程并以十六进制形式显示每个扇区的内容。使用 `iomani.setfill()` 在每个字节开头填充一个 0。

3. LongRandomArray 过程

以 12.4.3 节的 LongRandom 过程为起点，创建一个 LongRandomArray 过程，该过程以 32 位无符号随机整数填充数组。该过程要求从 C++ 中传递一个指向数组的指针以及一个表示要填充的数组元素数目的计数：

```
extern "C" void LongRandomArray( unsigned long * buffer,
                                unsigned count );
```

4. 外部 TranslateBuffer 过程

写一个与 12.2.2 节的内嵌 TranslateBuffer 过程执行同样加密功能的外部汇编语言过程。在调试器中运行编译后的程序，判断一下该版本是否比 12.2.2 节的 Encode.cpp 程序运行得更快。

5. 素数程序

写一个汇编语言过程，如果通过 EAX 传递的 32 位整数是素数则返回 1，否则返回 0。从高级语言中调用该过程，允许用户输入一些大数，程序为输入的每个数显示一条消息，说明其是否是素数。

6. FindRevArray 过程

修改 12.3.1 节的 FindArray 过程，把修改后的函数命名为 FindRevArray，使其从数组尾部开始搜索，反向查找第一个匹配项并返回匹配元素的指针值，如果没有匹配值则返回-1。

第 13 章 16 位 MS-DOS 程序设计

本章要点

- MS-DOS 和 IBM-PC
- MS-DOS 功能调用 (INT 21h)
- 标准 MS-DOS 文件 I/O 服务

13.1 MS-DOS 和 IBM-PC

在使用 Intel 8088 处理器的 IBM 个人计算机上,IBM 的 PC-DOS 是首个实地址模式操作系统,它后来演化成了 Microsoft 的 MS-DOS。由于这段历史,使用 MS-DOS 操作系统作为解释实地址模式程序设计的基本环境是很有意义的。实地址模式通常称为 16 位模式,因为该模式下的地址是由 16 位值构成的。

本章讲述 MS-DOS 的基本内存组织、如何使用 MS-DOS 功能调用(称为中断)以及如何在操作系统层次执行基本的输入输出操作。由于使用了 INT 指令,本章中的所有程序只能在实地址模式下运行。中断最早就是为实地址模式下的 MS-DOS 设计的,在保护模式下调用中断是可能的,但相关技术超出了本书的范围。

实地址模式程序具有以下特征:

- 只能寻址 1 MB 内存。
- 一次任务中只能运行一个程序(单任务)。
- 内存没有边界保护,因此任何应用程序都可以覆盖操作系统使用的内存。
- 偏移是 16 位的。

IBM-PC 在刚开始出现的时候呼声很高,其原因有二,首先是一般人买得起它,其次是它可以运行 Lotus 1-2-3 电子表格软件, Lotus 软件有助于 IBM-PC 为商业应用所接受。计算机爱好者喜欢 PC, 因为它是学习和了解计算机如何工作的理想工具。我们还应该注意一下 Digital Research 公司的 CP/M 操作系统, 这个在 PC-DOS 之前最流行的 8 位操作系统只能寻址 64 KB 的 RAM, 从这点看来, PC-DOS 能够寻址 640 KB 内存简直就是天赐之物。

由于早期 Intel 处理器在内存和速度上的明显限制, IBM-PC 只能是一个单用户的计算机。操作系统对应用程序造成的内存破坏没有任何内建的防护措施。相反, 当时的小型机能够处理多用户, 并能够防止应用程序之间相互覆盖数据。随着时间的推移, 用于 PC 的更健壮的操作系统出现了, 特别是在数台 PC 联网一起工作的时候, 其功能就更加强大, 这使得 PC 成为小型机之外的另外一种选择。

13.1.1 内存组织

在实地址模式下, 最低地址处的 640 KB 内存是由操作系统和应用程序共用的, 在此之上是为视频和硬件控制器保留的内存, 最高端的 F0000~FFFF 之间的内存是为系统 ROM(只读内存)保留的。图 13.1 是简单的内存映射图, 其中操作系统区域最低端的 1024 字节(00000~003FF)存放的是包含 32 位地址项的中断向量表(interrupt vector table), 这些 32 位地址称为中断向量, CPU 在处理硬件和软件中断时要使用这些中断向量。

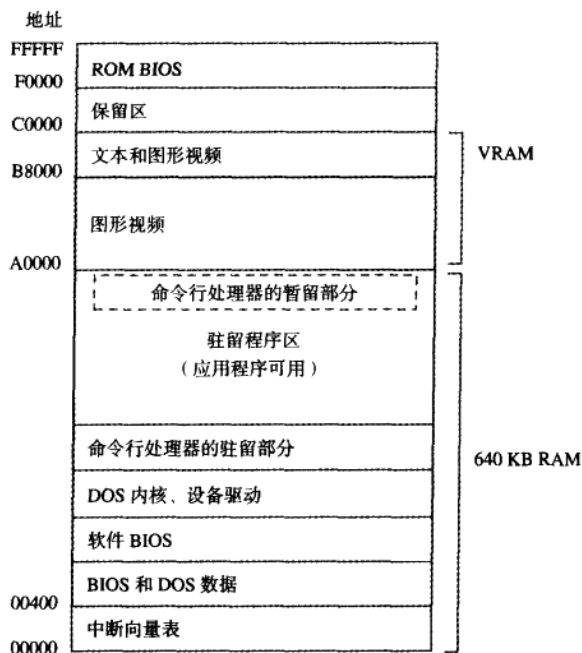


图 13.1 MS-DOS 内存映射

在中断向量表之上是 BIOS 和 MS-DOS 数据区，接下来是软件 BIOS 区，该区包含了管理键盘、磁盘、视频显示、串口和打印机口等大多数 I/O 设备的过程，BIOS 过程是从 MS-DOS 系统盘（引导盘）上的一个隐藏文件中装入的。MS-DOS 的内核是一系列过程（称为服务）的集合，这些过程也是从系统盘上的一个文件中装入的^①。

MS-DOS 内核区中包含了文件缓冲区和可安装的设备驱动程序，接下来的内存存放从可执行文件 `command.com` 中装入的命令处理器的驻留部分，命令行处理器解释在 MS-DOS 提示符下输入的命令，加载并执行存储在磁盘上的程序，命令行处理器的第二部分（暂留部分）存放在 A0000 之下的高端内存区。

应用程序可加载至命令行处理器驻留部分之上的最低可用地址中，可使用的内存地址范围最高可到 9FFFF。如果当前运行的程序覆盖了命令行处理器的暂留区，那么在程序退出时，系统将从引导盘上重新载入命令行处理器的暂留部分。

视频内存区（显存）：IBM-PC 的视频内存区（VRAM）从位置 A0000 开始，它在显示适配卡切换到图形模式时使用。内存位置 B8000 开始的内存区存放彩色文本模式下当前显示的所有字符。屏幕显示是该内存区的映射，屏幕上的每个坐标对应于映射内存中的一个 16 位的字，字符一旦被复制到缓冲区中，就会立即显示在屏幕上。

ROM BIOS：ROM BIOS 位于 F0000~FFFFF 之间的内存区中，这是计算机操作系统的一个重要区域，其中包含了系统诊断和配置软件以及应用程序使用的底层输入输出过程。BIOS 存储在系

① 这里的“软件 BIOS 区”和“BIOS 过程”指的是从系统盘中的 `IO.SYS` 文件中装入的子程序，而不是主板 ROM 中的 BIOS；MS-DOS 内核指的是从系统盘中的 `MSDOS.SYS` 文件中装入的子程序，本书对这方面的描述和国内大部分资料的描述在用语上有些不同——译者注。

统主板的静态内存芯片上。大多数系统都遵循一种标准的 BIOS 规范, 该规范是在 IBM 初始版本的 BIOS 之后标准化的, BIOS 使用的数据区位于 00400~004FF 之间。

13.1.2 重定向输入输出

本章中将用到标准输入设备 (standard input device) 和标准输出设备 (standard output device) 这两个名词。标准输入设备和标准输出设备统称为控制台 (console), 控制台一般从键盘读取输入, 使用视频显示进行输出。

在命令行上运行程序时可以重定向标准输入, 以便从文件或硬件端口输入而不是从键盘输入, 也可以重定向标准输出至文件、打印机或其他 I/O 设备中。如果没有重定向的能力, 改变输入输出之前就必须修改程序。例如, 操作系统有一个对输入文件进行排序的程序 sort.exe, 下面的命令对 myfile.txt 进行排序并显示输出:

```
sort < myfile.txt
```

下面的命令对 myfile.txt 进行排序并将输出送到 outfile.txt:

```
sort < myfile.txt > outfile.txt
```

读者可以使用管道符号 (|) 把 dir 命令的输出作为 sort.exe 的输入, 下面的命令对当前目录进行排序并在屏幕上显示结果:

```
dir | sort
```

下面的命令行把 sort 程序的输出送到默认 (非网络的) 的打印机 (以 prn 表示):

```
dir | sort > prn
```

标准设备名的完整列表如表 13.1 所示。

表 13.1 标准的 MS-DOS 设备名

设备名	描述
CON	控制台 (视频显示或键盘)
LPT1 或 PRN	第一个并口打印机
LPT2、LPT3	并口 2 和并口 3
COM1、COM2	串口 1 和串口 2
NUL	空设备

13.1.3 软件中断

软件中断 (software interrupt) 是对操作系统过程的调用。这些过程中的大多数被称为中断服务例程 (interrupt service routine) 或中断处理程序 (interrupt handler)。中断处理程序为应用程序提供了输入输出的能力, 通常用于类似如下的任务中:

- 显示字符和字符串
- 从键盘读取字符或字符串
- 显示彩色文本
- 打开和关闭文件
- 从文件读取数据
- 向文件写入数据
- 设置和返回系统的时间及日期

13.1.4 INT 指令

INT (中断过程调用) 指令在堆栈上压入 CPU 的标志并调用中断处理程序。在执行 INT 指令之前必须在寄存器中放入一个或多个参数, 最少也要在 AH 中存放一个标识需要调用的子过程的数字。根据调用功能的不同, 可能还要通过寄存器向中断传递其他参数值。中断指令的格式如下:

INT 中断号

其中的中断号是一个 0~FFh 之间的整数。

中断向量

CPU 使用中断向量表处理 INT 指令。前面曾提过, 中断向量表是存储在内存最低 1024 字节内的一张地址表。该表中的每一项都是一个 32 位的段-偏移形式的地址, 指向中断处理过程。表中地址的数值在不同的机器上并不相同。图 13.2 说明了在程序调用 INT 指令时 CPU 采取的动作。

- 步骤 1: INT 指令的操作数乘以 4, 以定位相应的中断向量表项。
- 步骤 2: CPU 在堆栈上压入标志寄存器、32 位的段-偏移返回地址, 禁止硬件中断并执行对存储在中断向量表中位置 (10h * 4) 处的地址 (F000:F065) 的调用。
- 步骤 3: F000:F065 处的中断处理程序开始执行, 直到执行 IRET 指令时结束。
- 步骤 4: IRET (中断返回) 指令从堆栈上弹出状态标志值和返回地址, 使 CPU 在调用程序中的 INT 10h 之后的下一条指令处恢复执行。

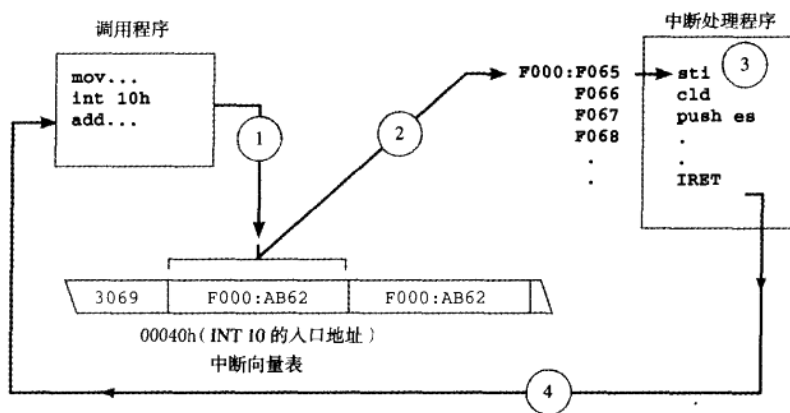


图 13.2 中断向量处理

常用的中断

软件中断调用 BIOS 或 DOS 的中断服务程序, 常用的中断包括:

- INT 10h 视频服务, 包括控制光标位置、显示彩色文本、滚动屏幕和显示图形等过程。
- INT 16h 键盘服务, 包括读取键盘和检查键盘状态的过程。
- INT 17h 打印服务, 包括初始化、打印和返回打印机状态的过程。
- INT 1Ah 时间服务, 包括获取自计算机启动之后的时钟滴答数以及刷新该计数器的过程。
- INT 1Ch 用户定时器中断服务, 每秒执行 18.2 次空白过程。
- INT 21h MS-DOS 服务, 提供输入输出、文件处理和内存管理等过程。也称为 MS-DOS 功能调用。

13.1.5 16 位程序的编写

为 MS-DOS 设计的程序必须是运行于实地址模式下的 16 位应用程序。实地址模式应用程序使用 16 位的段并使用 2.3.1 节描述的分段寻址方式。如果使用的是 32 位的处理器，即使是在实地址模式下，也可以使用 32 位的通用寄存器存放数据。下面是 16 位程序编码要点的小结。

- .MODEL 伪指令指定程序使用的内存模式。建议使用小型内存模式，在该模式下代码在一个单独的段内，数据和堆栈在另外一个段内：

```
.MODEL small
```

- .STACK 伪指令为程序分配一块局部堆栈空间。通常，只有在极少的情况下才需要超过 256 字节的堆栈空间，下面的例子为程序分配了 512 字节，已经非常宽裕了：

```
.STACK 200h
```

- 有时候，可能想允许使用 32 位寄存器。这可以通过使用 .386 伪指令做到：

```
.386
```

- 如果程序中引用了变量，在 main 的开始需要两条指令，初始化 DS 寄存器指向数据段的起始地址（MASM 预定义的常量 @data）：

```
mov ax,@data
mov ds,ax
```

- 每个程序必须包含结束程序并返回操作系统的语句，可以使用 .EXIT 伪指令：

```
.EXIT
```

- 此外，还可以调用 INT 21h 的功能 4Ch：

```
mov ah,4ch          ; 结束进程
int 21h             ; MS-DOS 中断
```

- 可使用 MOV 指令为段寄存器赋值，不过只应为段寄存器赋程序实际段的地址。
- 汇编 16 位程序时，使用 make16.bat（批处理）文件，该文件执行老版本（版本 5.6）的 16 位的 Microsoft 链接器并把程序同 Irvine16.lib 库链接。
- 运行于 MS-DOS，Windows 95/98/Me 下的实地址程序能够访问硬件端口、中断向量和系统内存，这类操作在 Windows NT/2000/XP 下是不允许的。
- 使用小型内存模式时，数据和代码标号的偏移地址是 16 位的。Irvine16 库使用小型内存模式，所有的代码都在一个 16 位的段内，所有的堆栈和数据都位于另外一个 16 位的段内。
- 在实地址模式下，堆栈项默认是 16 位的，不过仍然可以在堆栈上存放 32 位的值（使用两个堆栈项）。

包含 Irvine16.inc 文件能够简化 16 位程序的编码，Irvine16.inc 在程序的开头插入如下语句，定义了内存模式、调用约定、分配堆栈空间、允许使用 32 位寄存器并且把 .EXIT 伪指令重新定义成了 exit：

```
.MODEL small,stdcall
.STACK 200h
.386
exit EQU <.EXIT>
```

13.1.6 本节习题

1. 应用程序最高可加载到内存的哪个位置？

2. 占用内存最低处 1024 字节的是什么?
3. BIOS 和 MS-DOS 数据区的起始地址是多少?
4. 包含计算机使用的用于底层输入/输出过程的内存区是什么?
5. 给出一个把程序输出重定向到打印机的例子。
6. 第一个并行打印机的 MS-DOS 设备名是什么?
7. 什么是中断服务例程?
8. 当 INT 指令执行的时候, CPU 执行的第一个步骤是什么?
9. 程序调用 INT 指令的时候 CPU 执行的 4 个步骤是什么 (提示: 参考图 13.2)?
10. 中断服务例程结束时, 应用程序如何恢复执行?
11. 哪个中断号用于视频服务?
12. 哪个中断号用于时间服务?
13. 中断向量表中的哪个偏移地址包含了 INT 21h 中断处理程序的地址?

13.2 MS-DOS 功能调用 (INT 21h)

MS-DOS 提供了许多非常易于使用的功能以便在控制台上显示文本, 这些功能是 MS-DOS INT 21h 功能调用的一部分。INT 21h 支持 200 多个不同的功能调用, 这些调用通过在 AH 寄存器中放入的功能号进行区分。关于中断的一本优秀的但也许有点过时的教材是 Ray Duncan 的著作 *Advanced MS-DOS Programming* (高级 MS-DOS 程序设计)。一份更新更详尽的中断列表可在网上找到, 名字是 *Ralf Brown's Interrupt List* (Ralf Brown 中断列表), 细节请参阅本书的 Web 站点。

对于本章讲述的所有 INT 21h 中断功能, 我们都列出了必要的输入参数、返回值、使用注意事项以及一小段调用该功能的代码示例。

许多功能调用都要求 32 位的输入参数的偏移地址存放在 DS:DX 寄存器中。数据段寄存器 DS 通常指向程序的数据区。如果 DS 的值不符合实际的要求, 可以使用 SEG 操作符将其设置为包含要传递给 INT 21h 数据的段地址, 下面的语句就是这种情况。

```
.data
inBuffer BYTE 80 DUP(?)
.code
mov ax, SEG inBuffer
mov ds, ax
mov dx, OFFSET inBuffer
```

作者本人用 Intel 汇编语言编写的第一个程序 (大约在 1983 年) 是在屏幕上显示一个“*”:

```
mov ah, 2
mov dl, '*'
int 21h
```

有人说汇编语言非常难学, 但这个小程序使人大受鼓舞。正如事实所证明的, 在编写重要的应用之前, 还有一些细节需要学习。

INT 21h 功能 4Ch: 结束进程。INT 21h 功能 4Ch 结束当前的程序 (称为一个进程)。在本章给出的实地址模式程序中, 我们依赖于 Irvine16 库中的宏定义 `exit`, 它是这样定义的:

```
exit TEXTEQU <.EXIT>
```

换句话说, `exit` 是 `.EXIT` (一个结束程序的 MASM 伪指令) 的别名或替代, 使用 `exit` 符号的目的是为了仅使用这样的一条语句就可以结束 16 位或 32 位的程序。在 16 位程序中, `.EXIT` 生成的实

实际代码是：

```
mov ah,4Ch          ; 结束程序
int 21h
```

如果为.EXIT 宏提供了可选的返回码参数，汇编器将多生成一条指令来把返回码送到 AL 中：

```
.EXIT 0             ; 宏调用
```

这时生成的代码如下所示：

```
mov ah,4Ch          ; 结束程序
mov al,0             ; 返回码
int 21h
```

AL 中的值称为进程的返回值，它由调用程序（包括批处理文件）接收，用来表示程序的返回状态，一般约定用 0 表示成功完成，1~255 表示其他具有特定含义的返回值。例如，Microsoft 的汇编器 ML.EXE 在程序正确汇编时返回 0，否则返回非 0 值。

附录 C 包含了相当详尽的 BIOS 和 MS-DOS 中断列表。

13.2.1 精选的输出功能调用

本节给出了一些最常用的在控制台上显示字符和文本的 INT 21h 功能调用。这些功能调用都不能改变当前屏幕的默认颜色配置，因此如果前面通过其他方式设置了屏幕颜色（例如，可以调用第 15 章的视频 BIOS 功能调用），那么输出只能在已设置好的色彩下进行。

过滤控制字符：本小节中的所有的功能调用都过滤控制字符，或者说对 ASCII 控制字符进行解释。例如，如果向标准输出上写了一个退格符，光标就向左移动一格。表 13.2 列出了最可能遇到的控制字符。

表 13.2 ASCII 控制字符

ASCII 码	描 述
08h	退格（向左移动一列）
09h	水平制表符（向前跳过 n 列）
0Ah	换行符（移动到下一行）
0Ch	换页符（移动到下一页）
0Dh	回车符（移动到最左边一列）
1Bh	Escape 字符

下面的几张表描述了 INT 21h 的几个重要的功能调用：2、5、6、9 和 40h。INT 21h 功能 2 在标准输出上显示一个字符；INT 21h 功能 5 在打印机上打印一个字符；INT 21h 功能 6 在标准输出上显示一个未经过滤的字符；INT 21h 功能 9 在标准输出上显示字符串（以 \$ 字符结尾的）；INT 21h 功能 40h 把若干字节的数组写入文件或设备。

INT 21h 功能 2

描述	在标准输出上显示一个字符并把光标前进一个位置（一列）
接收参数	AH = 2 DL = 字符值
返回值	无
调用示例	mov ah,2 mov dl,'A' int 21h

INT 21h 功能 5

描述	在打印机上打印一个字符
接收参数	AH = 5 DL = 字符值
返回值	无
调用示例	<pre> mov ah,5 ; 选择打印机输出 mov dl,"Z" ; 要打印的字符 int 21h ; 调用 MS-DOS </pre>
注意	MS-DOS 一直等待直到打印机准备好为止。可按下 Ctrl-Break 键结束等待。默认输出到连接在 LPT1 端口上的打印机

INT 21h 功能 6

描述	在标准输出上显示一个字符
接收参数	AH = 6 DL = 字符值
返回值	如果 ZF = 0, AL 中包含字符的 ASCII 码
调用示例	<pre> mov ah,6 mov dl,"A" int 21h </pre>
注意	与其他 INT 21h 功能调用不同, 该功能调用不解释 ASCII 控制字符

INT 21h 功能 9

描述	在标准输出上显示以\$结尾的字符串
接收参数	AH = 9 DS:DX = 字符串的段-偏移地址
返回值	无
调用示例	<pre> .data string BYTE "This is a string\$" .code mov ah,9 mov dx,OFFSET string int 21h </pre>
注意	字符串必须以美元符号 (\$) 结尾

INT 21h 功能 40h

描述	把字节数组写入文件或设备
接收参数	AH = 40h BX = 文件或设备的句柄 (控制台 = 1) CX = 要写的字符数目 DS:DX = 数组的地址
返回值	AX = 已写的字节数

(续表)

INT 21h 功能 40h

调用示例

```
.data
message "Hello, world"
.code
mov ah,40h
mov bx,1
mov cx,LENGTHOF message
mov dx,OFFSET message
int 21h
```

13.2.2 例子程序: Hello World

下面是一个使用 MS-DOS 功能调用在屏幕上显示字符串的简单程序:

```
TITLE Hello World Program          (Hello.asm)

.MODEL small
.STACK 100h
.386

.data
message BYTE "Hello, world!",0dh,0ah
.code
main PROC
    mov ax,@data                ; 初始化 DS
    mov ds,ax

    mov ah,40h                  ; 写入文件/设备
    mov bx,1                    ; 输出句柄
    mov cx,SIZEOF message       ; 字节数
    mov dx,OFFSET message       ; 缓冲区的地址
    int 21h

    .EXIT
main ENDP
END main
```

另外一个版本: 编写 Hello.asm 的另外一种方式是使用预定义的.STARTUP 伪指令(该伪指令初始化 DS 寄存器), 这时要求删除 END 伪指令后的标号名:

```
TITLE Hello World Program          (Hello2.asm)

.MODEL small
.STACK 100h
.386

.data
message BYTE "Hello, world!",0dh,0ah
.code
main PROC
    .STARTUP

    mov ah,40h                  ; 写入文件/设备
    mov bx,1                    ; 输出句柄
    mov cx,SIZEOF message       ; 字节数
    mov dx,OFFSET message       ; 缓冲区的地址
    int 21h

    .EXIT
main ENDP
END
```

13.2.3 精选的输入功能调用

本节讲述一些最常用的从标准输入读入字符的 MS-DOS 功能调用。更完整的功能列表参见附录 C。如下表所示，INT 21h 的功能 1 从标准输入读入一个字符。

INT 21h 功能 1	
描述	从标准输入读取一个字符
接收参数	AH = 1
返回值	AL = 字符 (ASCII 码)
调用示例	<pre>mov ah,1 int 21h mov char,al</pre>
注意	如果输入缓冲区内无字符，则程序一直等待。该功能调用在标准输出上回显字符

如果输入缓冲区内有输入字符等待，INT 21h 功能 6 则读入一个字符。如果缓冲区为空，该功能返回并设置零标志，该功能调用不等待按键。

INT 21h 功能 6	
描述	从标准输入上读取一个字符，不等待
接收参数	AH = 6
	DL = FFh
返回值	如果 ZF = 0，AL 中存放着字符的 ASCII 码
调用示例	<pre>mov ah,6 mov dl,0FFh int 21h jz skip mov char,AL skip:</pre>
注意	只有在输入缓冲区内有字符等待时才能返回字符。不在标准输出上回显字符，也不过滤控制字符

INT 21h 功能 0Ah 从标准输入读取一个以回车字符结尾的字符串。如果调用了该功能，必须传递一个输入结构 (count 可以在 0~128 之间)，其格式如下：

```
count = 80
KEYBOARD STRUCT
    maxInput BYTE count           ; 最多可输入的字符数
    inputCount BYTE ?            ; 实际输入的字符数
    buffer BYTE count DUP(?)     ; 存放输入的字符
KEYBOARD ENDS
```

maxInput 域指定了用户可最多输入的字符数 (包括回车键在内)。退格键可用来删除字符并回移光标。用户可按下回车键或 Ctrl-Break 组合键结束输入。PageUp 和 F1 等非 ASCII 键都被过滤掉，不会存储在缓冲区之内。在该功能调用返回之后，inputCount 域表明输入了多少字符 (回车键不包括在内)。下表对功能 0Ah 进行了说明。

INT 21h 功能 0Ah	
描述	从标准输入读取缓冲字符数组
接收参数	AH = 0Ah
	DS:DX = 键盘输入结构的地址

选择 239 作为加密值完全是随意的, 这里可以使用 0~255 之间的任意值, 不过, 如果使用 0 的话是不会起到任何加密的作用的。当然, 这种加密是很脆弱的, 但已经足够使得试图攻破该加密的普通用户丧失信心。在命令行下运行程序时, 可标明输入文件 (和输出文件, 如果有的话) 的名字。下面是两个例子:

encrypt < infile.txt	从文件输入 (infile.txt), 输出到控制台
encrypt < infile.txt > outfile.txt	从文件输入 (infile.txt), 输出到文件 (outfile.txt)

INT 21h 功能 3Fh

如下表所示, INT 21h 功能 3Fh 从文件或设备读取一个字节数组。在 BX 中的设备句柄等于 0 时可用于从键盘输入。

INT 21h 功能 3Fh	
描述	从文件或设备读取字节数组
接收参数	AH = 3Fh BX = 文件/设备句柄 (0 = 键盘句柄) CX = 最多读取的字符数 DS:DX = 输入缓冲区的地址
返回值	实际读取的字节数
调用示例	.data inputBuffer BYTE 127 dup(0) bytesRead WORD ? .code mov ah, 3Fh mov bx, 0 mov cx, 127 mov dx, OFFSET inputBuffer int 21h mov bytesRead, ax
注意	如果从键盘读, 在按下回车键时输入终止, 0Dh 和 0Ah 被追加到输入缓冲区的末尾

如果用户输入的字符比功能调用要求得多, 那么多余的字符仍然保留在输入缓冲区中。如果程序后面运行中的其他时刻又调用了该功能, 程序或许不会停止以等待输入, 因为缓冲区中可能已经包含了数据 (包括标记行结束的 0Dh, 0Ah)。这种情况甚至可能会在不同的程序间发生。为了确保程序按编写者的意图运行, 在调用功能 3Fh 之后需要一次一个字符地清除缓冲区。下面的代码可以做到这一点 (完整的说明参见 Keybd.asm 程序)。

```

;-----
FlushBuffer PROC
; Flush the standard input buffer.
; Receives: nothing. Returns: nothing
;-----
.data
oneByte BYTE ?
.code
pusha
L1:
mov ah, 3Fh          ; 读取文件/设备
mov bx, 0            ; 键盘句柄

```

```

mov cx,1                ; 一个字节
mov dx,OFFSET oneByte   ; 保存在这里
int 21h                 ; 调用 MS-DOS
cmp oneByte,0Ah         ; 行已结束?
jne L1                  ; 否: 读取下一个字符
popa
ret
FlushBuffer ENDP

```

13.2.4 日期/时间功能调用

许多流行的软件都显示当前的日期和时间, 还有一些软件返回日期和时间供程序内部使用。例如, 一个日程规划程序会使用当前日期以确保用户不会安排一个已经过去的约会。

如下面的表格所示, INT 21h 功能 2Ah 获取系统日期, INT 21h 功能 2Bh 设置系统日期; INT 21h 功能 2Ch 获取系统时间, INT 21h 功能 2Dh 设置系统时间。

INT 21h 功能 2Ah	
描述	获取系统日期
接收参数	AH = 2Ah
返回值	CX = 年 DH, DL = 月、日 AL = 星期日期 (星期日=0, 星期一 = 1, 等)
调用示例	<pre> mov ah,2Ah int 21h mov year,cx mov month,dh mov day,dl mov dayOfWeek,al </pre>

INT 21h 功能 2Bh	
描述	设置系统日期
接收参数	AH = 2Bh CX = 年 DH = 月 DL = 日
返回值	如果设置成功, AL = 0, 否则 AL = FFh
调用示例	<pre> mov ah,2Bh mov cx,year mov dh,month mov dl,day int 21h cmp al,0 jne failed </pre>
注意	在 Windows NT/2000/XP 下以受限用户身份运行时可能不能正常工作

INT 21h 功能 2Ch

描述	获取系统时间
接收参数	AH = 2Ch
返回值	CH = 小时数 (0~23) CL = 分钟数 (0~59) DH = 秒数 (0~59) DL = 百秒数 (一般不是很精确)
调用示例	mov ah,2Ch int 21h mov hours,ch mov minutes,cl mov seconds,dh

INT 21h 功能 2Dh

描述	设置系统时间
接收参数	AH = 2Dh CH = 小时数 (0~23) CL = 分钟数 (0~59) DH = 秒数 (0~59)
返回值	如果设置成功, AL = 0, 否则 AL = FFh
调用示例	mov ah,2Dh mov ch,hours mov cl,minutes mov dh,seconds int 21h cmp al,0 jne failed
注意	在 Windows NT/2000/XP 下以受限用户身份运行时可能不能正常工作

例子: 显示时间和日期

下面的程序 (DateTime.asm) 显示系统日期和时间, 程序的代码比想像的要长一点, 因为程序用了一些额外的代码在小时、分钟和秒之前插入 0:

```
TITLE Display the Date and Time      (DateTime.asm)

Include Irvine16.inc
Write PROTO char:BYTE
.data
str1 BYTE "Date: ",0
str2 BYTE "  Time: ",0
.code
main PROC
    mov ax,@data
    mov ds,ax

; 显示日期
    mov dx,OFFSET str1
    call WriteString
    mov ah,2Ah                ; 获取系统日期
```

```

    int    21h
    movzx  eax,dh                ; 月
    call   WriteDec
    INVOKE Write,'-'
    movzx  eax,dl                ; 日
    call   WriteDec
    INVOKE Write,'-'
    movzx  eax,cx                ; 年
    call   WriteDec

; 显示时间
    mov    dx,OFFSET str2
    call   WriteString
    mov    ah,2Ch                ; 获取系统时间
    int    21h
    movzx  eax,ch                ; 小时
    call   WritePaddedDec
    INVOKE Write,':'
    movzx  eax,cl                ; 分钟
    call   WritePaddedDec
    INVOKE Write,':'
    movzx  eax,dh                ; 秒
    call   WritePaddedDec
    call   Crlf

    exit
main ENDP

;-----
Write PROC char:BYTE
; Display a single character.
;-----
    push  eax
    push  edx
    mov   ah,2                ; 字符显示功能调用
    mov   dl,char
    int   21h
    pop   edx
    pop   eax
    ret
Write ENDP

;-----
WritePaddedDec PROC
; Display unsigned integer in EAX, padding
; to two digit positions with a leading zero.
;-----
    .IF  eax < 10
        push  eax
        push  edx
        mov   ah,2                ; 显示开头的 0
        mov   dl,'0'
        int   21h
        pop   edx
        pop   eax
    .ENDIF

```

```
    call WriteDec          ; 显示无符号整数
    ret                   ; 使用 EAX 中的值
WritePaddedDec ENDP
END main
```

输出示例:

```
Date: 12-8-2001, Time: 23:01:23
```

13.2.5 本节习题

1. 调用 INT 21h 时哪个寄存器存放功能调用号?
2. INT 21h 的哪个功能调用终止程序?
3. INT 21h 的哪个功能调用在标准输出上显示一个字符?
4. INT 21h 的哪个功能调用在标准输出上显示一个以\$结尾的字符串?
5. INT 21h 的哪个功能调用向文件和设备写一块数据?
6. INT 21h 的哪个功能调用从标准输入上读取一个字符?
7. INT 21h 的哪个功能调用从标准输入设备上读取一块数据?
8. 如果想要获取系统日期然后显示, 再改变日期, 需要哪些 INT 21h 功能调用?
9. INT 21h 的哪些功能调用在 Windows NT/2000/XP 下以受限用户身份运行时可能不能正常工作?
10. 应该使用 INT 21h 的哪个功能调用检查标准输入缓冲区中是否有字符等待处理?

13.3 标准 MS-DOS 文件 I/O 服务

INT 21h 提供了数量众多的文件和目录 I/O 服务, 在这里全部详细地讲述是不可能的, 表 13.3 中列出了一些读者可能会用到的功能调用。

表 13.3 INT 21h 文件和目录相关功能

功 能	描 述
716Ch	创建或打开文件
3Eh	关闭文件句柄
42h	移动文件指针
5706h	获取文件创建的日期和时间

文件/设备句柄: MS-DOS 和 MS-Windows 使用句柄 (handle, 一般是一个 16 位的整数) 来标识文件和 I/O 设备。有 5 个预定义的设备句柄, 除了句柄 2 (错误输出) 之外的其他句柄都支持命令行重定向, 这些句柄在任何时刻都是可用的:

- 0 键盘 (标准输入)
- 1 控制台 (标准输出)
- 2 错误输出
- 3 辅助设备 (异步)
- 4 打印机

所有 I/O 功能调用都有一个共同的特点: 如果失败则设置进位标志, 并在 AX 中返回出错码。读者可以根据出错码来显示合适的提示信息。表 13.4 列出了出错码及其含义。

表 13.4 MS-DOS 扩展错误码

错 误 码	含 义
01	无效功能号
02	文件未找到
03	路径未找到
04	打开文件太多(无剩余句柄)
05	访问无效
06	无效句柄
07	内存控制块被破坏
08	内存不足
09	无效内存块地址
0A	无效的环境
0B	无效的格式
0C	无效访问码
0D	无效的数据
0E	保留
0F	指定了无效的驱动器
10	试图删除当前目录
11	非同一设备
12	无更多文件
13	磁盘写保护
14	未知单元
15	驱动器未准备好
16	未知命令
17	数据错误(CRC)
18	无效的请求结构长度
19	读/写指针移动错误
1A	未知媒体类型
1B	扇区未找到
1C	打印机缺纸
1D	写错误
1E	读错误
1F	一般错误

13.3.1 创建或打开文件(716Ch)

INT 21h 功能调用 716Ch 创建新文件或打开已存在的文件, 该功能允许使用扩展文件名及文件共享属性, 文件名中可包含目录路径。该功能调用描述如下所示。

INT 21h 功能 716Ch

描述	创建新文件或打开已存在的文件
接收参数	AX=716Ch BX= 访问模式 (0= 读, 1= 写, 2= 读/写) CX= 属性 (0= 普通, 1=只读, 2= 隐藏, 3= 系统, 8= 卷标, 20h= 存档) DX= 动作 (1= 打开, 2= 剪裁, 10h= 创建) DS:SI= 文件名的段/偏移地址 DI= 别名提示 (可选)
返回值	如果创建/打开成功, CF = 0, AX = 文件句柄, CX = 采取的动作。如果创建/打开失败, CF = 1
调用示例	<pre> mov ax,716Ch ; 扩展打开/创建 mov bx,0 ; 只读 mov cx,0 ; 普通属性 mov dx,1 ; 打开已存在的文件 mov si,OFFSET Filename int 21h jc failed mov handle,ax ; 文件句柄 mov actionTaken,cx ; 采取的动作 </pre>
注意	BX 中的访问模式还可以和下面的共享模式值联合使用: OPEN_ SHARE_ COMPATIBLE, OPEN_ SHARE_ DENYREADWRITE, OPEN_ SHARE_ DENYWRITE, OPEN_ SHARE_ DENYREAD, OPEN_ SHARE_ DENYNONE。CX 中返回的动作值可以是: ACTION_OPENED, ACTION_CREATED_ OPENED, ACTION_ REPLACED_OPENED。所有这些常量值都在 Irvine16.inc 中定义

更多例子：下面的代码创建一个新文件或剪裁^①一个已存在的同名文件：

```

mov ax,716Ch          ; 扩展打开/创建
mov bx,2              ; 读写
mov cx,0              ; 普通属性
mov dx,10h + 02h      ; 动作：创建+剪裁
mov si,OFFSET Filename
int 21h
jc failed
mov handle,ax         ; 文件句柄
mov actionTaken,cx    ; 打开文件时采取的动作

```

下面的代码试图创建一个新文件，如果文件已存在则失败（设置进位标志）：

```

mov ax,716Ch          ; 扩展打开/创建
mov bx,2              ; 读写
mov cx,0              ; 普通属性
mov dx,10h            ; 动作：创建
mov si,OFFSET Filename
int 21h
jc failed
mov handle,ax         ; 文件句柄
mov actionTaken,cx    ; 打开文件时采取的动作

```

① 即打开并将其中的内容清空——译者注。

13.3.2 关闭文件句柄(3Eh)

INT 21h 功能调用 3Eh 关闭一个文件的句柄。这个功能调用刷新文件的写缓冲区, 把任何残留数据写入磁盘, 如下表所示。

INT 21h 功能 3Eh	
描述	关闭文件句柄
接收参数	AH = 3Eh BX = 文件句柄
返回值	如果文件成功关闭, CF = 0, 否则 CF = 1
调用示例	<pre>.data filehandle WORD ? .code mov ah,3Eh mov bx,filehandle int 21h jc failed</pre>
注意	如果文件被修改, 其时间戳和日期戳也被更新

13.3.3 移动文件指针(42h)

INT 21h 功能调用 42h 把已打开文件的指针移动到新的位置。当调用该功能时, AL 中存放的方式代码标识了应该如何设置指针:

- 0 相对于文件头开始的偏移
- 1 相对于当前位置开始的偏移
- 2 相对于文件末尾开始的偏移

INT 21h 功能 42h	
描述	移动文件指针
接收参数	AH = 42h AL = 方式代码 BX = 文件句柄 CX:DX = 32 位的偏移值
返回值	如果移动文件指针成功, CF = 0 并且 DX:AX 中返回新的文件指针, 否则 CF = 1
调用示例	<pre>mov ah,42h mov al,0 ; 方法: 相对于开始的偏移地址 mov bx,handle mov cx,offsetHi mov dx,offsetLo int 21h</pre>
注意	DX:AX 中返回的文件指针偏移总是相对于文件开始的

13.3.4 获取文件创建的日期和时间

INT 21h 功能 5706h 获取文件创建的日期和时间, 该时间和文件最后被修改或访问的日期和时间不一定相同, 如下表所示。如果想要了解 MS-DOS 时间和日期的压缩格式, 请参见 14.3.1 节; 析取日期/时间域的例子请参见 7.3.4 节。

INT 21h 功能 5706h

描述	获取文件创建的日期和时间
接收参数	AX = 5706h BX = 文件句柄
返回值	如果功能调用成功, CF = 0, DX = 日期 (MS-DOS 压缩格式), CX = 时间, SI = 毫秒。 如果功能调用失败, CF = 1
调用示例	<pre> mov ax,5706h ; 获取创建日期/时间 mov bx,handle int 21h jc error ; 失败则退出 mov date,dx mov time,cx mov milliseconds,si </pre>
注意	文件必须已经被打开。毫秒值标识是要加到MS-DOS时间上的以 10 ms 为单位的间隔数, 范围从 0~199, 这说明该域最多可以使总时间增加 2 s

13.3.5 精选的库例程

这里列举了 Irvine16 链接库中的两个过程: ReadString 和 WriteString。ReadString 的编写是比较需要技巧的, 因为它必须一次读取一个字符, 直到遇到行结束符 (0Dh) 为止。该函数从标准输入上读取这个字符, 但并不把它存入缓冲区。

ReadString

ReadString 过程从标准输入读取一个字符串并把字符串放在一个输入缓冲区中, 以空字符结尾, 该过程在用户按下回车键时结束。

```

;-----
ReadString PROC
; Receives: DS:DX points to the input buffer,
;           CX = maximum input size
; Returns:  AX = size of the input string
; Comments: Stops when the Enter key (0Dh) is pressed.
;-----
    push cx                ; 保存寄存器
    push si
    push cx                ; 再次保存数字计数器
    mov si,dx              ; 指向输入缓冲区
L1: mov ah,1               ; 功能: 键盘输入
    int 21h                ; 在 AL 中返回字符
    cmp al,0Dh              ; 行已经结束?
    je L2                  ; 是: 退出
    mov [si],al             ; 否: 存储字符
    inc si                  ; 缓冲区指针加 1
    loop L1                 ; 循环直到 CX=0
L2: mov byte ptr [si],0     ; 以 NULL 字符结尾
    pop ax                  ; 原始的数字计数器
    sub ax,cx               ; AX = 输入字符串的长度
    pop si                  ; 恢复寄存器
    pop cx
    ret
ReadString ENDP

```

WriteString

WriteString 过程在标准输出上显示一个以空字符结尾的字符串。它调用一个辅助过程 Str_length 来返回字符串的长度。

```

;-----
WriteString PROC
; Writes a null-terminated string to standard output
; Receives: DS:DX = address of string
; Returns: nothing
;-----
    pusha
    push ds                      ; 设置 ES 等于 DS
    pop  es
    mov  di,dx                   ; ES:DI = 字符串的指针
    call Str_length              ; AX = 字符串长度
    mov  cx,ax                   ; CX = 字节数
    mov  ah,40h                  ; 写入文件或设备
    mov  bx,1                    ; 标准输出句柄
    int  21h                     ; 调用 MS-DOS
    popa
    ret
WriteString ENDP

```

13.3.6 例子：读取并复制文本文件

本章前面在讲述如何读取标准输入时介绍过 INT 21h 功能调用 3Fh，如果 BX 中的文件句柄代表一个已经打开用于输入的文件，则该功能也可以用于读文件。当功能 3Fh 返回时，AX 中返回实际从文件中读取的字节数。当到达文件末尾时，AX 中返回的字节数比请求读取的字节数（在 CX 中）要少。

同样，在本章前面讲述向标准输出写的时候（设备句柄 1）提到了 INT 21h 功能 40h，该功能调用 BX 中的句柄也可以是一个已打开的文件句柄。该功能调用自动更新文件的位置指针，因此再次调用功能 40h 时将从上次写入后的位置开始继续写。

下面演示的 Readfile.asm 程序解释了本节讲述的几个 INT 21h 功能调用的用法：

- 功能 716Ch，创建新文件或打开已存在的文件
- 功能 3Fh，从文件或设备读
- 功能 40h，向文件或设备写
- 功能 3Eh，关闭文件句柄

下面的程序打开一个文本文件用于输入，从文件读取最多 5000 个字节然后在控制台上显示，同时创建一个新文件并把数据复制到新文件中：

```

TITLE Read a text file          (Readfile.asm)

; Read, display, and copy a text file.
INCLUDE Irvine16.inc

.data
BufSize = 5000
infile   BYTE "my_text_file.txt",0
outfile  BYTE "my_output_file.txt",0
inHandle WORD ?

```



```

outHandle WORD ?
buffer      BYTE BufSize DUP(?)
bytesRead  WORD ?

.code
main PROC
    mov ax,@data
    mov ds,ax
; 打开输入文件
    mov ax,716Ch                ; 扩展的创建或打开
    mov bx,0                    ; 模式 = 只读
    mov cx,0                    ; 普通属性
    mov dx,1                    ; 动作: 打开
    mov si,OFFSET infile
    int 21h                     ; 调用 MS-DOS
    jc quit                     ; 如果发生错误则退出
    mov inHandle,ax

; 读取输入文件
    mov ah,3Fh                  ; 读取文件或设备
    mov bx,inHandle             ; 文件句柄
    mov cx,BufSize              ; 最多读取的字节数
    mov dx,OFFSET buffer        ; 缓冲区指针
    int 21h
    jc quit                     ; 如果发生错误则退出
    mov bytesRead,ax

; 显示缓冲区
    mov ah,40h                  ; 写文件或设备
    mov bx,1                    ; 控制台输出句柄
    mov cx,bytesRead            ; 字节数
    mov dx,OFFSET buffer        ; 缓冲区指针
    int 21h
    jc quit                     ; 如果发生错误则退出

; 关闭文件
    mov ah,3Eh                  ; 功能: 关闭文件
    mov bx,inHandle             ; 输入文件句柄
    int 21h                     ; 调用 MS-DOS
    jc quit                     ; 如果发生错误则退出

; 创建输出文件
    mov ax,716Ch                ; 扩展的创建或打开
    mov bx,1                    ; 模式 = 只写
    mov cx,0                    ; 普通属性
    mov dx,12h                  ; 动作: 创建/剪裁
    mov si,OFFSET outfile
    int 21h                     ; 调用 MS-DOS
    jc quit                     ; 如果发生错误则退出
    mov outHandle,ax            ; 保存句柄

; 把缓冲区写入一个新文件
    mov ah,40h                  ; 写文件或设备
    mov bx,outHandle            ; 输出文件句柄
    mov cx,bytesRead            ; 字节数
    mov dx,OFFSET buffer        ; 缓冲区指针
    int 21h

```

```

        jc  quit                                ; 如果发生错误则退出
; 关闭文件
        mov  ah,3Eh                            ; 功能: 关闭文件
        mov  bx,outHandle                      ; 输出文件句柄
        int  21h                              ; 调用 MS-DOS

quit:
        call Crlf
        exit
main ENDP
END main

```

13.3.7 读取 MS-DOS 命令行

我们经常用命令行向程序传递信息,假设我们需要向程序 `attr.exe` 传递文件名 `file1.doc`,相应的 MS-DOS 命令行应该是:

```
attr FILE1.DOC
```

程序启动时,命令行上的任何文本都自动存储在一个 128 字节的 MS-DOS 命令行区域中,该命令行区域位于由 `ES` 寻址的段内偏移 `80h` 处。这块内存区域称为程序段前缀 (PSP, Program Segment Prefix),相关内容在 16.3.1 节讲述,还可参考 2.3.1 节对实地址模式下分段寻址如何工作的讨论。

命令行区域中的第一个字节是命令行的长度,如果长度大于 0,那么第二个字节是一个空格,再后面跟命令行上实际键入的字符。以 `attr.exe` 为例,用十六进制格式表示的命令行内容如下所示:

偏移:	80	81	82	83	84	85	86	87	88	89	8A	8B
内容:	0A	20	46	49	4C	45	31	2E	44	4F	43	0D
	F I L E 1 . D O C											

如果在使用 CodeView 调试器加载程序运行之前设置了命令行参数,可以在 CodeView 中看到命令行字节。

要在 CodeView 中设置命令行参数,选择 **Run** 菜单下的 **Runtime Arguments** 选项。要查看参数,可以按下 **F10** 键执行程序的第一条指令,然后打开一个内存窗口,从 **Options** 菜单中选择 **Memory** 选项,在 **Address Expression** 域中键入 `ES:0x80`。

MS-DOS 存储命令或文件名后的所有字符的规则有一种例外情况:使用重定向输入输出时,MS-DOS 不保存使用的文件和设备名。例如,在键入下面的命令时,MS-DOS 不保存任何东西,因为 `infile.txt` 和 `PRN` 都是用于重定向的:

```
progl < infile.txt > prn
```

GetCommandTail 过程: 库 `Irvine16` 中的 `GetCommandTail` 过程返回在 MS-DOS 下运行的程序的命令行的一份副本。调用该过程时把 `DX` 设置为接收命令行的缓冲区地址。实地址模式程序通常直接使用段寄存器前缀以访问不同段内的数据。例如, `GetCommandTail` 在堆栈上保存 `ES` 的当前值,然后使用 `INT 21h` 功能调用 `62h` 获取 PSP 所在的段并把该值复制到 `ES` 中:

```

push es
.
.
mov  ah,62h                                ; 获取 PSP 的段地址

```

```
int 21h          ; 在 BX 中返回
mov es,bx        ; 复制到 ES 中
```

接下来, 该过程定位 PSP 内的一个字节, 由于 ES 不指向程序的默认数据段, 因此必须使用段超越前缀 (es:) 寻址 PSP 内的数据:

```
mov cl,es:[di-1] ; 获取长度字节
```

过程使用 SCASB 指令跳过开头的空格, 如果命令行为空则设置进位标志, 这样如果命令行上未键入任何东西, 调用程序就可以很容易地用一条 JC 指令 (进位则跳转) 来检测到这种情况:

```
cld          ; 向前扫描
mov al,20h   ; 空格字符
repz scasb   ; 扫描非空格字符
jz L2        ; 所有空格都已找到
.
.
L2: stc      ; CF=1 意味着无命令行
```

SCASB 指令扫描的内存是由 ES 段寄存器寻址的, 因此别无选择, 只能在 GetComm and Tail 过程的开始处把 ES 设置为 PSP 的段地址。下面是该过程的完整清单:

```
GetCommandTail PROC
;
; Gets a copy of the MS-DOS command tail at PSP:80h.
; Receives: DX contains the offset of the buffer
; that receives a copy of the command tail.
; Returns: CF=1 if the buffer is empty; otherwise,
; CF=0.
;-----
SPACE = 20h
push es
pusha          ; 保存通用寄存器
mov ah,62h     ; 获取 PSP 的段地址
int 21h        ; 在 BX 中返回
mov es,bx      ; 复制到 ES 中

mov si,dx      ; 缓冲区指针
mov di,81h     ; 命令行在 PSP 中偏移地址
mov cx,0       ; 字节计数
mov cl,es:[di-1] ; 获取长度字节
cmp cx,0       ; 命令行为空?
je L2          ; 是: 退出
cld            ; 向前扫描
mov al,SPACE   ; 空格字符
repz scasb     ; 扫描非空格字符
jz L2          ; 所有空格都已找到
dec di        ; 未找到空格
inc cx

```

默认情况下, DI 是 DS 中保存的段内的偏移, 段超越前缀 (es:[di]) 通知寄存器在寻址 [DI] 时使用 ES 中保存的段。

```
L1: mov al,es:[di] ; 复制命令行至缓冲区
    mov [si],al    ; 由 DS:SI 指向
    inc si
    inc di
    loop L1
    clc            ; CF=0 意味着找到了命令行
    jmp L3

```

```

L2: stc                                ; CF=1 意味着无命令行
L3: mov     byte ptr [si],0           ; 存储 null 字节
    popa                                ; 恢复寄存器
    pop     es
    ret
GetCommandTail ENDP

```

13.3.8 例子：创建二进制文件

二进制文件的得名是由于存储在文件中的数据只不过是程序数据的映像。例如，假设你的程序创建并填充了一个双字数组：

```
myArray DWORD 50 DUP(?)
```

如果想要把这个数组写到一个文本文件中，就不得不把每个整数转换成字符串并分别单独写入。一种更加高效的方式是把 myArray 的二进制映像写入一个文件，50 个双字的数组使用 200 字节的内存，这也就是文件将占用的磁盘空间的数量。

下面的 Binfile.asm 程序用随机整数填充一个数组，并在屏幕上显示，然后把整数写入二进制文件，最后关闭文件。接下来程序又重新打开文件，读取整数并在屏幕上显示：

```

TITLE Binary File Program              (Binfile.asm)

; This program creates a binary file containing
; an array of doublewords. It then reads the file
; back in and displays the values.
INCLUDE Irvine16.inc
.data
myArray DWORD 50 DUP(?)
fileName BYTE "binary array file.bin",0
fileHandle WORD ?
commaStr  BYTE ", ",0

; Set CreateFile to zero if you just want to
; read and display the existing binary file.
CreateFile = 1

.code
main PROC
    mov ax,@data
    mov ds,ax

    .IF CreateFile EQ 1
        call FillTheArray
        call DisplayTheArray
        call CreateTheFile
        call WaitMsg
        call Crlf
    .ENDIF
    call ReadTheFile
    call DisplayTheArray

quit:
    call Crlf
    exit
main ENDP
;-----
ReadTheFile PROC

```

```

;
; Open and read the binary file.
; Receives: nothing.
; Returns: nothing
;-----
    mov ax,716Ch          ; 扩展的文件打开
    mov bx,0              ; 模式: 只读
    mov cx,0              ; 属性: 普通
    mov dx,1              ; 打开已存在的文件
    mov si,OFFSET fileName ; 文件名
    int 21h               ; 调用 MS-DOS
    jc quit               ; 如果发生错误则退出
    mov fileHandle,ax      ; 保存句柄

; 读取输入文件, 然后关闭文件
    mov ah,3Fh            ; 读文件或设备
    mov bx,fileHandle      ; 文件句柄
    mov cx,SIZEOF myArray ; 最多读取的字节数
    mov dx,OFFSET myArray ; 缓冲区指针
    int 21h
    jc quit               ; 输出文件句柄
    mov ah,3Eh            ; 功能: 关闭文件
    mov bx,fileHandle      ; 如果发生错误则退出
    int 21h               ; 调用 MS-DOS

quit:
    ret
ReadTheFile ENDP

;-----
DisplayTheArray PROC
;
; Display the doubleword array.
; Receives: nothing.
; Returns: nothing
;-----
    mov cx,LENGTHOF myArray
    mov si,0

L1:
    mov eax,myArray[si]    ; 取一个数字
    call WriteHex          ; 显示数字
    mov edx,OFFSET commaStr ; 显示一个逗号
    call WriteString
    add si,TYPE myArray    ; 下一个数组位置
    loop L1
    ret
DisplayTheArray ENDP

;-----
FillTheArray PROC
;
; Fill the array with random integers.
; Receives: nothing.
; Returns: nothing
;-----
    mov CX,LENGTHOF myArray

```

```

        mov     si,0
L1:
        mov     eax,1000                ; 生成随机整数
        call    RandomRange             ; 在 EAX 中, 范围是 0~999
        mov     myArray[si],eax         ; 存储在数组中
        add     si,TYPE myArray         ; 下一个数组位置
        loop    L1
        ret
FillTheArray ENDP
;-----
CreateTheFile PROC
;
; Create a file containing binary data.
; Receives: nothing.
; Returns: nothing
;-----
        mov     ax,716Ch                ; 创建文件
        mov     bx,1                    ; 模式: 只写
        mov     cx,0                    ; 普通文件
        mov     dx,12h                  ; 动作: 创建/剪裁
        mov     si,OFFSET fileName      ; 文件名
        int     21h                     ; 调用 MS-DOS
        jc      quit                    ; 如果发生错误则退出
        mov     fileHandle,ax           ; 保存句柄

; 向文件写入整型数组
        mov     ah,40h                  ; 写文件或设备
        mov     bx,fileHandle           ; 输出文件句柄
        mov     cx,SIZEOF myArray       ; 字节数
        mov     dx,OFFSET myArray       ; 缓冲区指针
        int     21h
        jc      quit                    ; 如果发生错误则退出

; 关闭文件
        mov     ah,3Eh                  ; 功能: 关闭文件
        mov     bx,fileHandle           ; 输出文件句柄
        int     21h                     ; 调用 MS-DOS

quit:
        ret
CreateTheFile ENDP
END main

```

值得注意的是, 把整个数组写入文件只需调用一次 INT 21h 的功能调用 40h 即可完成。无须使用循环:

```

mov ah,40h                ; 写文件或设备
mov bx,fileHandle         ; 输出文件句柄
mov cx,SIZEOF myArray     ; 字节数
mov dx,OFFSET myArray     ; 缓冲区指针
int 21h

```

把文件读回数组时也是如此, 一条对 INT 21h 功能 3Fh 的调用就能完成全部工作:

```

mov ah,3Fh                ; 读文件或设备
mov bx,fileHandle         ; 文件句柄
mov cx,SIZEOF myArray     ; 最多读取的字节数
mov dx,OFFSET myArray     ; 缓冲区指针
int 21h

```

13.3.9 本节习题

1. 说出 5 种标准的 MS-DOS 设备句柄的名字。
2. 在调用 MS-DOS I/O 功能之后，哪个标志表示发生了错误？
3. 在调用 716Ch 创建文件的时候需要哪些参数？
4. 给出一个打开已存在的文件用于输入的例子。
5. 调用 716Ch 从一个已打开的文件中读取一个二进制数组的时候需要哪些参数？
6. 使用 INT 21h 功能 3Fh 读输入文件时如何检查文件的结尾？
7. 当调用 3Fh 时，从文件中读和从键盘读有什么不同？
8. 如果想要读取一个随机文件，INT 21h 的哪个功能允许直接跳转到文件中间的某条特定记录处？
9. 写一小段代码，把文件指针定位在距文件开始 50 字节处（假设文件已经被打开，BX 中存放的是文件句柄）。

13.4 本章小结

本章讲解了 MS-DOS 的基本内存组织、如何激活 MS-DOS 功能调用和如何在操作系统层次执行基本的输入/输出操作等内容。

标准输入设备和标准输出设备统称为控制台，一般用键盘作为输入设备，使用视频显示器作为输出设备。

软件中断是对操作系统过程的调用，大多数操作系统过程被称为中断处理过程，它们为应用程序提供了输入输出的能力。

INT（中断过程调用）指令在堆栈上压入 CPU 标志和 32 位的返回地址（CS 和 IP），禁用其他中断并调用中断处理过程。CPU 使用中断向量表处理 INT 指令。中断向量表包含了 32 位的中断处理程序的段-偏移地址。

为 MS-DOS 设计的程序必须是运行于实地址模式下的 16 位应用程序。实地址模式程序使用 16 位的段并使用分段寻址方式。

.MODEL 伪指令指定程序使用何种内存模式，.STACK 伪指令为程序分配一块堆栈空间。在实地址模式下，堆栈项默认是 16 位的，可使用 .386 伪指令以允许使用 32 位的寄存器。

包含变量的 16 位应用程序在访问变量之前必须把 DS 设置为数据段的段值。

每个程序都必须包含一条结束程序并返回到操作系统的语句。一种方法是使用.EXIT 伪指令，另外一种方法是调用 INT 21h 的 4Ch 功能。

运行于 MS-DOS、Windows 95/98/Me 下时，任何实地址模式程序都可以访问硬件端口、中断向量以及系统内存。在 Windows NT/200/XP 下，这类操作只限于内核和设备驱动程序。

程序运行时，命令行上任何多余的文本都自动存储在程序段前缀（PSP）80h 处的 MS-DOS 命令行区域中，该区域的长度是 128 字节。Irvine16 库中的 GetCommandTail 过程返回命令行的一份副本。程序段前缀将在 16.3.1 节介绍。

一些常用的 BIOS 中断如下所示：

- INT 10h 视频服务，包含控制光标位置、写彩色文本、滚动屏幕和显示图形的过程。
- INT 16h 键盘服务，包含读取键盘状态和检查其状态的过程。
- INT 17h 打印机服务，包含初始化、打印和返回打印机状态的过程。

- INT 1Ah 每天的时间, 获取自机器启动后的时钟滴答数或将计数设置为新值。
- INT 1Ch 用户时钟中断, 每秒执行 18.2 次的空过程。

INT 21h MS-DOS 服务提供了输入输出、文件处理和内存管理的过程。INT 21h 通常被称为 MS-DOS 功能调用。该中断支持大约 200 个不同的功能, 不同的功能由 AH 寄存器中存放的功能号来标识。本章讲述了许多重要的 INT 21h 功能调用:

- INT 21h 功能调用 4Ch 终止当前的程序(称为进程)。
- INT 21h 功能调用 2 和 6 在标准输出上显示一个字符。
- INT 21h 功能调用 5 在打印机上打印一个字符。
- INT 21h 功能调用 9 在标准输出上显示一个字符串。
- INT 21h 功能调用 40h 向文件或设备写一个字节数组。
- INT 21h 功能调用 1 从标准输入上读取一个字符。
- INT 21h 功能调用 6 从标准输入上读取一个字符(不进行等待)。
- INT 21h 功能调用 0Ah 从标准输入读取缓冲字符串。
- INT 21h 功能调用 0Bh 获取标准输入缓冲区的状态。
- INT 21h 功能调用 3Fh 从文件或设备中读取一个字节数组。
- INT 21h 功能调用 2Ah 获取系统日期。
- INT 21h 功能调用 2Bh 设置系统日期。
- INT 21h 功能调用 2Ch 获取系统时间。
- INT 21h 功能调用 2Dh 设置系统时间。
- INT 21h 功能调用 716Ch 创建一个文件或打开一个已存文件。
- INT 21h 功能调用 3Eh 关闭文件的句柄。
- INT 21h 功能调用 42h 移动文件指针位置。
- INT 21h 功能调用 5706h 获取文件创建的日期和时间。
- INT 21h 功能调用 62h 返回程序段前缀的段地址。

下面的例子程序演示了如何使用 MS-DOS 功能调用:

- DateTime.asm 程序显示系统的日期和时间。
- Readfile.asm 程序打开一个文本文件用于输入, 在控制台上显示, 创建一个新文件并把数据复制到新文件中。
- Binfile.asm 程序用随机整数填充数组, 并在屏幕上显示, 然后把整数写入一个二进制文件, 最后关闭文件。接下来重新打开文件来读取整数并在屏幕上显示。

二进制文件的得名是由于存储在这种文件中的只是程序数据的映像。

13.5 练习题

下面的练习必须在实模式下完成。不要使用 Irvine16 库中的任何函数。除非特别说明, 请使用 INT 21h 的功能调用进行输入输出。

1. 读一个文本文件

打开一个文件用于输入, 读取该文件, 并在屏幕上以十六进制显示其内容。编写时使输入缓冲区较小(大约 256 字节), 用循环重复调用功能 3Fh 直到整个文件处理完毕。

2. 复制一个文本文件

修改 13.3.6 节的 Readfile 程序，以便能够读取任意大小的文件，假设缓冲区比输入文件小，使用一个循环来读取所有数据。如果在任何一次 INT 21h 功能调用之后进位标志置位，显示适当的错误信息。

3. 设置日期

写一个程序，显示当前的日期并提示用户输入一个新的日期。如果输入的日期非空，则使用这个值更新系统日期。

4. 大写转换

写一个程序，使用 INT 21h 从键盘输入小写字母并转换成大写字母，只显示大写字母。

5. 文件创建日期

写一个过程，显示文件的创建日期及文件名，使用 DX 传递文件名的偏移地址。写一个测试程序，以几个不同的文件名（包括扩展名）解释说明该过程的使用法。如果无法找到文件，显示一条适当的错误提示信息。

6. 文本匹配程序

写一个程序，可打开最大 60 KB 的文本文件并搜索字符串，搜索是大小写敏感的。字符串和文件名由用户输入。显示字符串在文件中出现的所有行，在每行前面显示行号。回顾一下 9.7 节中的 Str_find 过程，但是注意这个程序必须在实地址模式下运行。

7. 使用 XOR 加密文件

按以下步骤增强 6.3.4 节的文件加密程序：

- 提示用户输入明文文件名和加密文本文件名。
- 打开纯文本文件用于输入，并且打开加密文本文件用于输出。
- 让用户输入一个整数加密代码（0~255）。
- 把输入文件读入缓冲区，使用加密代码异或每个字节。
- 把缓冲区写入加密文本文件中。

唯一可以使用的过程是本书链接库中的 ReadInt 函数，所有其他的输入输出功能必须用 INT 21h 来完成。同样的代码也能够用于解密加密文件产生明文文件。

8. CountWords 过程

写一个程序，计算文本文件中的单词数。提示用户输入文件名，并在屏幕上显示单词数目，唯一可以调用的过程是本书链接库中的 WriteDec 函数，所有其他输入输出必须使用 INT 21h 来完成。

第 14 章 磁盘基础知识

本章要点

- 磁盘存储系统
- 文件系统
- 磁盘目录
- 读写磁盘扇区（7305h）
- 系统级文件功能调用

14.1 磁盘存储系统

本章介绍磁盘存储系统的基础知识。在介绍了基于 Intel 的计算机上物理磁盘存储和 BIOS 级的存储接口之间的关系之后，讲述 MS-Windows 是如何同应用程序进行交互以提供对文件和目录的访问的。系统 BIOS（基本输入输出系统）在 2.5 节已经介绍过了。在考察磁盘存储的时候，可以看到计算机各虚拟层之间的交互关系是相当明显的（参见图 14.1）：

- 最底层是磁盘控制器固件，它使用智能控制芯片把对不同品牌的磁盘的逻辑操作转换为对物理磁盘的读写。
- 再上一层是系统 BIOS，提供了若干底层调用供操作系统使用，以执行扇区读写、磁道格式化等任务。
- 最上层是操作系统 API（应用编程接口），包含了一系列提供磁盘服务的函数，以实现如打开和关闭文件、设置文件属性、读写文件等功能。

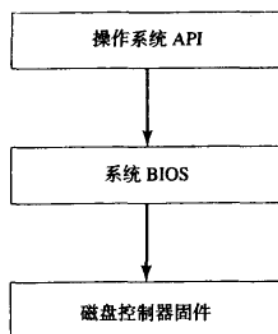


图 14.1 磁盘访问的虚拟层次

所有的磁盘存储系统都有一些共同点：它们自动处理数据的物理分区，在文件层访问数据并且把文件名映射到物理存储。在硬件层，磁盘存储以盘片、面、磁道、柱面和扇区进行描述；在系统 BIOS 层，磁盘存储以簇和扇区进行描述；在操作系统层，磁盘存储以文件和目录进行描述。

汇编语言程序：在 MS-DOS，Windows 95/98/Me 下，汇编语言编写的用户级程序可以直接访

问系统 BIOS，这可能非常有用——读者或许想以非常格式存取数据，或许想恢复丢失的数据，或者想对磁盘硬件进行诊断。本章给出了一些使用系统 BIOS 功能调用进行文件和扇区操作的例子。为了说明典型的操作系统级数据访问，在本章的最后还介绍了一些用于磁盘和目录操作的 MS-DOS 功能调用。

如果读者使用的是 Windows NT/2000/XP，用户级的应用程序只能使用 Win32 API 访问磁盘系统。这是为了确保系统安全，只有运行在最高特权级的设备驱动程序才能绕过这种限制。

14.1.1 磁道、柱面和扇区

典型的硬盘结构如图 14.2 所示，它由固定在以一定速度旋转的轴上的多个盘片组成，每个盘片的表面都有用来记录磁脉冲的读写磁头。读写磁头以组为单位向盘片的中央或边缘步进。

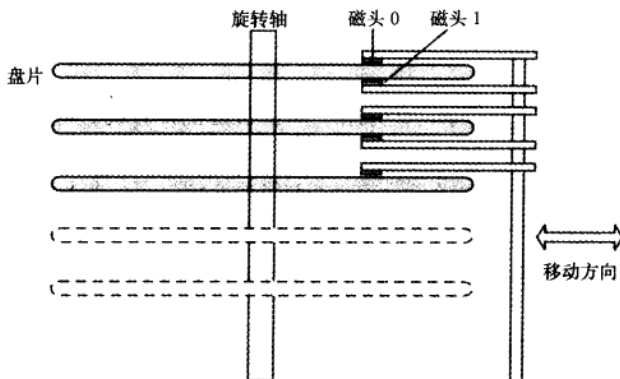


图 14.2 硬盘的物理构成

磁盘的表面被格式化成肉眼无法看见的同心圆，称为磁道（track），数据在磁道上以磁记录的方式存储，典型的 3.5 英寸（1 英寸 = 2.54 厘米）硬盘包含上万条磁道。把读写磁头从一个磁道移到另一个磁道称为寻道，平均寻道时间是磁盘速度的一项测量指标。磁盘速度的另外一个指标是每分钟转数（r/min），典型的磁盘转速是 7200 r/min。最外圈磁道的序号是 0，磁头向中央移动时磁道号是递增的。

柱面（cylinder）是指读写磁头在某位置时可访问的所有磁道。开始时磁盘上的文件总是存储在相邻的柱面上的，这样可以减少磁头的移动。

扇区（sector）是磁道上按每 512 字节划分的块，如图 14.3 所示。物理扇区是由制造商使用一种称为低级格式化（low-level format）的手段所做的磁性标记（不可见）。不管安装的是什么操作系统，扇区的大小是永远不会改变的。在硬盘中每磁道可包含 63 个或者更多的扇区。

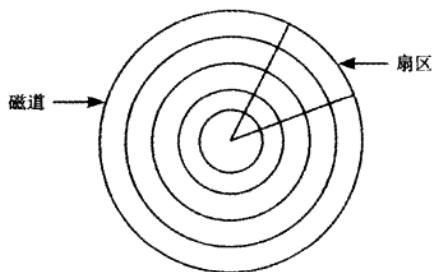


图 14.3 磁盘的磁道和扇区

磁盘的物理参数用于描述磁盘结构, 以使其能够为系统 BIOS 理解和读取。磁盘的物理参数由每盘片柱面数、每柱面的读写磁头数和每道扇区数构成。存在如下关系:

- 每盘片柱面数等于每个盘片表面的磁道数。
- 总磁道数等于柱面数与每柱面读写磁头数的乘积。

碎片:随着时间的流逝, 硬盘上的文件数量越来越多, 被不断写入和删除, 碎片就会产生。碎片 (fragment) 文件是指其存储扇区在磁盘上并不连续的文件。在这种情况下, 读取文件数据时磁头就不得不跨越磁道了, 这将降低文件的读写速度。

逻辑扇区号的转换:硬盘控制器执行一个转换过程, 以把磁盘的物理参数转换成能被操作系统理解的逻辑结构。控制器通常嵌入于固件 (firmware) 中, 它安装在硬盘内或者位于一块独立的控制卡上。在转换之后, 操作系统就可以使用逻辑扇区号 (logical sector number) 进行操作了。逻辑扇区号总是连续的, 从 0 开始。

14.1.2 磁盘分区 (卷)

在 MS-Windows 下, 单个物理硬盘可分成一个或多个称为分区 (partition) 或卷 (volume) 的逻辑部分。每个格式化的分区由一个独立的驱动器符代表, 如 C、D、E 等, 对磁盘进行格式化时可以使用几种文件系统中的任意一种。驱动器中可以包含两种类型的分区: 主分区和扩展分区。对磁盘进行分区时可以采用下面两种方案, 具体取决于是否需要扩展分区。

主分区通常用于存放操作系统, 是可引导的。扩展分区可分成任意数目的逻辑分区, 每个逻辑分区都以独立的驱动器符的形式出现。逻辑分区不可引导。格式化每个系统或逻辑分区时使用不同的文件系统是允许的。

以一个 20 GB 的硬盘为例, 假设主分区是 10 GB (驱动器 C), 里面安装操作系统。扩展分区也是 10 GB, 可以把扩展分区随意地分成 2 GB 和 8 GB 的两个逻辑分区, 并用 FAT16, FAT32, NTFS 等不同文件系统对它们进行格式化 (这些文件系统的细节将在下一节中讨论)。如果没有安装其他硬盘, 这两个逻辑分区将被分配驱动器符 D 和 E。

多引导系统:创建多个主分区以引导不同的操作系统的做法是非常普遍的, 这使得在不同的环境下测试软件并利用高级系统的安全特性成为可能。很多软件开发者使用一个主分区来为开发中的软件创建测试环境, 使用其他主分区来存放已经测试完毕并准备交付给消费者使用的软件产品。

相反, 逻辑分区通常用于存放数据。不同的操作系统共享存储在同一分区上的数据是可能的, 例如, 较新版本的 MS-Windows 和 Linux 都可以读取 FAT32 分区, 计算机可以从任何一种操作系统引导并读取存储在共享逻辑分区上的数据。

工具:可使用 MS-DOS 和 Windows 98 下的 FDISK.EXE 实用程序创建和删除分区, 但是 FDISK 并不保留原来的数据。Windows 2000 和 Windows XP 提供了一个更好的磁盘管理实用工具, 可在不破坏数据的情况下创建、删除和重新划分分区的大小。一些第三方的软件, 如 Symantec 公司的 PartitionMagic 允许在不破坏数据的情况下移动分区或重新划分分区大小。

双重引导的例子:在图 14.4 中, Windows 2000 磁盘管理工具显示了一个硬盘上所有的 6 个分区。图中显示的系统可从 Windows 98 或 Windows 2000 引导, 其中有两个主分区, 分别命名为 SYSTEM 98 和 WIN2000-A, 但同一时刻只能有一个活动分区。活动的主分区也称为系统分区。

Volume	Layout	Type	File System	Status	Capacity	Free Space	% Free
	Partition	Basic		Healthy	5.13 GB	5.13 GB	100 %
	Partition	Basic		Healthy	2.01 GB	2.01 GB	100 %
BACKUP (E:)	Partition	Basic	FAT32	Healthy	7.80 GB	4.84 GB	62 %
DATA_1 (D:)	Partition	Basic	FAT32	Healthy	7.80 GB	2.66 GB	34 %
SYSTEM 98	Partition	Basic	FAT32	Healthy	1.95 GB	1.12 GB	57 %
WIN2000-A (C:)	Partition	Basic	NTFS	Healthy (System)	3.91 GB	1.43 GB	36 %

图 14.4 Windows 2000 磁盘管理工具

图中当前的系统分区是 WIN2000-A, 分配给它的盘符是 C。注意非活动的系统分区没有驱动器符, 如果重启计算机以 SYSTEM 98 分区引导, 它将变成驱动器 C, WIN2000-A 分区将变成非活动分区。

同时, 扩展分区被分成 4 个逻辑分区, 其中的两个尚未格式化, 另两个使用 FAT32 文件系统格式化并被命名为 BACKUP 和 DATA_1。

主引导记录: 主引导记录 (MBR, Master Boot Record) 位于硬盘的第一个逻辑分区中, 它是在创建第一个硬盘分区时创建的, 其中包含了以下信息:

- 硬盘分区表, 描述了磁盘上所有分区的位置和大小。
- 一段小程序, 用于定位分区引导扇区并把控制权转移给分区引导扇区中的代码, 由后者负责加载操作系统。

14.1.3 本节习题

1. (对/错) 磁道被分成多个称为扇区的逻辑单位。
2. (对/错) 扇区由多个磁道构成。
3. _____是由在某位置的所有读写磁头能够访问的所有磁道构成的。
4. (对/错) 物理扇区的大小总是 512 字节, 因为扇区是由制造商在磁盘上标记的。
5. 在 FAT32 文件系统下, 一个逻辑扇区使用多少字节?
6. 为什么开始时文件保存在相邻的柱面上?
7. 文件存储开始碎片化后, 这对于硬盘的柱面和寻道操作而言意味着什么?
8. 硬盘分区的另一个名字是_____。
9. 硬盘的平均寻道时间测量的是什么?
10. 什么是低级格式化?
11. 主引导记录里面包含什么?
12. 同一时刻可以有多少个活动的主分区?
13. 如果主分区是活动的, 就称为_____分区。

14.2 文件系统

每个操作系统都有某种形式的磁盘管理系统。磁盘管理系统在最底层管理分区, 在上一层管理文件和目录。文件系统必须跟踪每个磁盘文件的位置、大小和属性。让我们看看 FAT 类型的文件系统, FAT 文件系统最初是为 IBM-PC 设计的, 如今 Windows 系统仍然在使用。FAT 类型的文件系统提供了下面的结构:

- 把逻辑扇区映射为簇 (cluster), 簇是所有文件和目录的基本存储单位。
- 把文件和目录名映射为簇序列。

簇是文件使用空间的最小单位，它包含一个或多个相邻的磁盘扇区。文件系统将每个文件存储为簇的链表序列。簇的大小依赖于使用的文件系统的类型以及磁盘分区的大小。图 14.5 显示了一个由两个 2048 字节的簇组成的文件，其中每个簇包含 4 个 512 字节的扇区。簇链可以在文件分配表（FAT）中查找，后者跟踪文件使用的所有簇。指向 FAT 表中文件的首个簇的指针存储在每个文件的目录项中。14.3.3 节更详细地解释了 FAT 的结构。

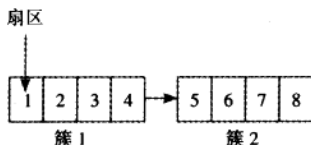


图 14.5 簇链表的例子

浪费的空间：即使是小文件，也要求使用至少一个磁盘簇来存储，这会导致空间的浪费。图 14.6 显示了一个 8200 字节的文件在完全填满两个 4096 字节的簇之后，还使用了第三个簇的 8 个字节，这在第三个簇中留下了 4088 字节浪费的空间。实际上，4096 字节大小的簇（4 KB）被认为是存储小文件的有效方式。设想一下，如果 8200 字节的文件存储在 32 KB 簇的分区上会发生什么，在这种情况下会导致 24 568（32 768 - 8200）字节的浪费。在有大量小文件的分区上，使用小尺寸的簇是最好的。

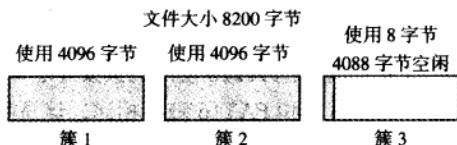


图 14.6 显示浪费空间情况的簇链表示意图

Windows 2000/XP 的例子：硬盘使用的各种文件系统类型和标准的簇大小如表 14.1 所示。随着新的操作系统的发布，这些值经常会发生变化，因此表中的信息可能很快就会过时。

表 14.1 分区和簇的大小（大于 1 GB）

分区大小	FAT16 簇	FAT32 簇	NTFS 簇 ^a
1.25 GB~2 GB	32 KB	4 KB	2 KB
2 GB~4 GB	64 KB ^b	4 KB	4 KB
4 GB~8 GB	ns（表示不支持）	4 KB	4 KB
8 GB~16 GB	ns	8 KB	4 KB
16 GB~32 GB	ns	16 KB	4 KB
32 GB~2 TB	ns	ns ^c	4 KB

a NTFS 下的默认大小，在格式化时其值可变。

b 只有 Windows 2000 和 XP 才支持 64 KB 簇的 FAT16 文件系统。

c 有一个软件补丁允许 Windows 98 格式化大于 32 GB 的硬盘。

14.2.1 FAT12

FAT12 文件系统最先用在 IBM-PC 的磁盘上，所有版本的 MS-Windows 和 Linux 到现在为止还支持这种格式，FAT12 文件系统中簇的大小为 512 字节，因此对于存储小文件是非常理想的。其文件分配表的每个项只有 12 位长，因此一个 FAT12 分区包含的簇不能超过 4087 个。

14.2.2 FAT16

在 MS-DOS 下, FAT16 文件系统是在格式化硬盘时唯一可用的文件系统。所有的 MS-Windows 和 Linux 都支持 FAT16 文件系统, FAT16 文件系统有一些缺点:

- 在大于 1 GB 的分区上存储效率是很低的, 因为 FAT16 使用大尺寸的簇。
- 文件分配表的每个表项是 16 位长的, 这限制了簇的总数目。
- 每个卷可包含簇的数目在 4087~65526 之间。
- 引导扇区没有备份, 因此单个扇区的读错误可能是灾难性的。
- 没有内建的文件系统安全属性或独立的用户访问许可限制。

14.2.3 FAT32

FAT32 文件系统是随着 Windows 95 OEM2 的发布而引入的, 并在 Windows 98 下做了进一步的改进。与 FAT16 相比, FAT32 文件系统有了很大的进步:

- 单个文件的大小最大可达 4 GB 减 2 字节。
- 文件分配表中每个表项的长度是 32 位的。
- 一个分区可以有 65 526~268 435 456 个簇。
- 根目录可位于磁盘上的任意位置, 可以是任意大小。
- 分区最大可达 32 GB。
- 在 1 GB ~8 GB 的分区上使用的簇比 FAT16 使用的簇要小, 减少了空间的浪费。
- 引导记录包含了关键数据的备份。和 FAT16 磁盘相比, 这意味着 FAT32 不容易受单点故障的影响。

14.2.4 NTFS

NTFS 文件系统被 Windows NT/2000/XP 所支持, 它较 FAT32 有重大的改进:

- NTFS 可处理非常大的卷, 一个卷既可以在一个硬盘上也可以跨越多个硬盘。
- 大于 2 GB 的盘的默认簇大小是 4 KB。
- 支持最多 255 个字符长的 Unicode (非 ANSI) 文件名。
- 可以设置文件和目录的访问许可限制, 允许单个用户或组用户的访问, 不同种类的访问 (读、写和修改等) 也是允许的。
- 对文件、目录和卷提供内建的加密和压缩。
- 可在更改日志中记录文件的变化。
- 可为单个用户和组用户设置磁盘配额 (限额)。
- 提供了健壮的数据错误恢复能力, 通过跟踪事务记录自动修复错误。
- 支持磁盘镜像, 同样的数据可同时写到多个驱动器上。

表 14.2 列出了基于 Intel 处理器的计算机常用的文件系统, 该表还显示了各种操作系统对各种文件系统的支持情况。

表 14.2 操作系统对文件系统的支持

文件系统	MS-DOS	Linux	Win95/98	WinNT 4	Win2000/XP
FAT12	支持	支持	支持	支持	支持

(续表)

文件系统	MS-DOS	Linux	Win95/98	WinNT 4	Win2000/XP
FAT16	支持	支持	支持	支持	支持
FAT32		支持	支持		支持
NTFS				支持	支持

14.2.5 磁盘的基本区域

FAT12 和 FAT16 卷都为引导记录、文件分配表和根目录区保留了特定的位置 (FAT32 分区的根目录位置并不固定)。每个区域的大小是在格式化卷的时候确定的。例如, 一张 3.5 英寸 (1.44 MB) 软盘的扇区映射如表 14.3 所示。

表 14.3 1.44 MB 软盘的扇区映射

逻辑扇区	内 容
0	引导记录
1~18	文件分配表 (FAT)
19~32	根目录
33~2879	数据区

引导记录: 引导记录 (boot record) 包含一张存储卷信息的表和一段把 MS-DOS 装入内存的小引导程序, 引导程序的作用是检查是否存在特定的操作系统文件并将其装入内存。表 14.4 列出了典型的 MS-DOS 引导记录中的各个域。应该注意的是, 各个域的实际组织方式在不同版本的操作系统之间是不同的。

表 14.4 MS-DOS 引导记录的布局

偏 移	长 度	描 述
00	3	跳转到引导代码 (JMP 指令)
03	8	制造商名称, 版本号
0B	2	每扇区字节数
0D	1	每簇扇区数 (2 的幂)
0E	2	(在 FAT #1 之前的) 保留扇区数
10	1	FAT 的数目
11	2	最多根目录项的数目
13	2	小于 32 MB 盘的扇区数
15	1	介质描述字节
16	2	以扇区数为单位的 FAT 的大小
18	2	每道扇区数
1A	2	磁头数
1C	4	隐藏扇区数
20	4	大于 32 MB 盘的扇区数
24	1	驱动器号 (由 MS-DOS 修改)
25	1	保留

(续表)

偏 移	长 度	描 述
26	1	扩展引导标志 (总是 29h)
27	4	卷的 ID 号 (二进制)
2B	11	卷标号
36	8	文件系统类型 (ASCII)
3E	—	引导程序和数据的开始

文件分配表 (FAT): 文件分配表相当复杂, 因此将在 14.3.3 节再详细讨论。

根目录 (root directory): 根目录是一个磁盘卷的主目录。根目录的每个表项既可以包含其他目录名也可以包含文件描述表项, 文件描述表项中包含了文件名、文件大小、属性、文件使用的起始簇号等信息。

数据区 (data area): 数据区是存储文件和子目录的地方。

14.2.6 本节习题

1. (对/错) 文件系统把逻辑扇区映射为簇。
2. (对/错) 文件的起始簇号存储在磁盘参数表中。
3. (对/错) 除 NTFS 以外的所有文件系统都要求使用至少一个簇来存储文件。
4. (对/错) FAT32 文件系统允许设置单个用户的目录访问许可, 但是不允许设置文件访问许可。
5. (对/错) Linux 不支持 FAT32 文件系统。
6. 在 Windows 98 下, FAT16 分区 (卷) 允许的最大容量是多少?
7. 假设磁盘卷的引导记录损坏了, 哪种文件系统支持引导记录备份?
8. 哪种文件系统支持 16 位的 Unicode 文件名?
9. 哪种文件系统支持磁盘镜像, 也就是说同样的数据可同时写入多个硬盘?
10. 假设需要跟踪文件的最后 10 次修改, 哪种文件系统支持该特性?
11. 如果有一个 20 GB 的磁盘卷, 并且准备使用小于等于 8 KB 的簇 (避免浪费空间), 可以使用哪些文件系统?
12. 支持 4 KB 簇的 FAT32 磁盘卷的最大容量是多大?
13. 按顺序描述一下 1.44 MB 软盘的 4 个区域。
14. 在 MS-DOS 格式化的驱动器上, 应如何确定每簇包含多少个扇区?
15. 挑战: 如果一个磁盘的簇的大小是 8 KB, 存储 8200 字节的文件时浪费的空间是多少?
16. 挑战: 解释 NTFS 系统如何存储稀疏文件 (回答该问题需要访问 Microsoft 的 MSDN 站点查阅相关信息)。

14.3 磁盘目录

每个 FAT 或 NTFS 格式的磁盘都有一个根目录, 根目录中包含了磁盘上文件的列表。根目录中还可以包含其他目录, 后者称为子目录 (subdirectory)。子目录可以看做是出现在其他目录中的目录——后者被称为父目录 (parent directory)。每个子目录都可以包含文件名和其他目录名, 结果就形成了一个树状结构, 根目录在最顶端, 下面的分支是子目录。整个结构如图 14.7 所示。

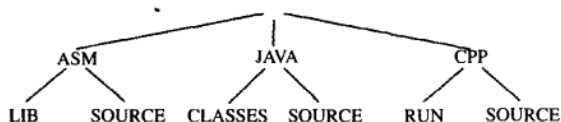


图 14.7 磁盘目录树的例子

一个目录中的每个目录和文件的名称都是由其上层的目录名来修饰的，通常被称为路径（path）。例如，ASM 目录下 SOURCE 目录中的 PROG1.ASM 文件的路径是：

```
C:\ASM\SOURCE\PROG1.ASM
```

在当前驱动器中进行输入输出操作时，驱动器符通常可以省略。上面目录树例子的完整目录名列表如下：

```
C:\
\ASM
\ASM\LIB
\ASM\SOURCE
\JAVA
\JAVA\CLASSES
\JAVA\SOURCE
\CPP
\CPP\RUN
\CPP\SOURCE
```

文件名的格式可采用单独的文件名或目录路径后跟文件名的形式，前面也可以加驱动器符。

14.3.1 MS-DOS 目录结构

如果要讲解当今基于 Intel 处理器的计算机的所有目录格式，必须至少包括 Linux、MS-DOS 和所有版本的 MS-Windows。不过，本章首先使用 MS-DOS 作为最基本的例子并仔细研究其目录结构，然后再讲解 MS-Windows 下使用的扩展文件结构。

每个 MS-DOS 目录项都是 32 字节长的，包含的各个域如表 14.5 所示。文件名域用于存放文件、目录或卷标的名字。文件名域的字节表示文件名第一个字母或文件的状态，其可能的状态值如表 14.6 所示。16 位的起始簇号域是指定了分配给文件的第一个簇的簇号，同时也指定了文件在文件分配表中的起始项。文件大小域表示按字节计算的文件大小，这是一个 32 位数。

表 14.5 MS-DOS 目录项

十六进制偏移	文件名	格 式
00~07	文件名	ASCII
08~0A	扩展名	ASCII
0B	属性	8 位二进制值
0C~15	MS-DOS 保留	
16~17	时间戳	16 位二进制值
18~19	日期戳	16 位二进制值
1A~1B	起始簇号	16 位二进制值
1C~1F	文件大小	32 位二进制值

表 14.6 文件名状态字节

状态字节	描 述
00h	该目录项未使用
01h	如果属性字节等于 0Fh 并且状态字节等于 01h, 表示该目录项是第一个长文件名目录项
05h	文件名的第一个字节实际上是 E5h (很少见)
E5h	目录项包含一个文件名, 但是该文件已被删除
2Eh	代表当前目录, 如果第二个字节也是 2Eh, 则簇号域包含该目录的父目录的簇号
4nh	第一个长文件名目录项, 如果属性值字节是 0Fh, 表示这是包含长文件名的多个目录项中的最后一个目录项。数字 n 表示长文件名使用的目录项数目

属性域

属性域指示文件的类型。该域是位映射的, 通常可包含如图 14.8 所示值的组合。两个保留位应永远置 0, 修改文件时设置存档位。如果目录项包含子目录名, 则子目录位置位。卷标表示该目录项用于表示磁盘卷的名字。系统文件位表示该文件是操作系统的一部分。隐藏文件位使文件隐藏, 文件名将不出现在目录列表中。只读位防止以任何方式删除或修改文件。最后, 属性值 0Fh 表示当前目录项用于扩展的长文件名。

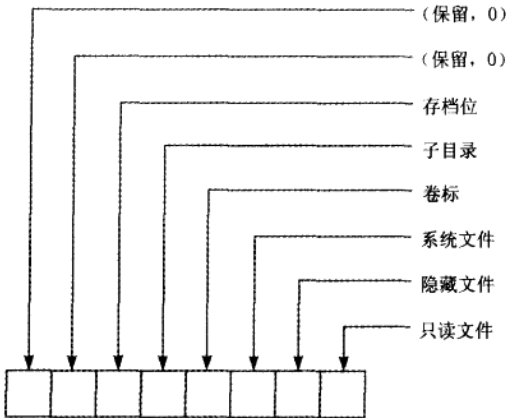


图 14.8 文件属性域字节

日期和时间

日期戳域（参见图 14.9）表示文件创建或最后修改的日期，它是以位映射值表示的。年的取值范围在 0~119 之间，该值要和 1980 年（IBM-PC 发布的那一年）相加以获得正确的时间。月的取值范围在 0~12 之间，日期的取值范围在 1~31 之间。

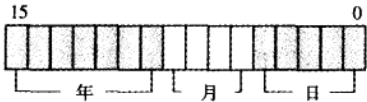


图 14.9 文件的日期戳域

时间戳域（参见图 14.10）表示文件被创建或最后修改的时间，该域也是以位映射值方式表示的。小时的取值范围是 0~23，分钟的取值范围是 0~59，秒计数的取值范围是 0~29，也就是说每个计数单位表示 2 秒的增量。例如二进制值 10100 等于 40 秒, 图 14.11 所表示的时间是 14:02:40。

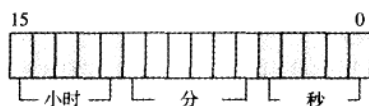


图 14.10 文件的时间戳域

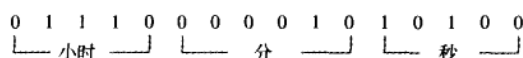


图 14.11 时间戳域的例子

文件目录项的例子：让我们仔细看一下图 14.12 所示的 MAIN.CPP 文件的目录项：文件有标准属性并且存档位（20h）已置位，这表示文件已经被修改过了；起始簇号是 0020h，大小是 000004EEh 字节；时间戳域等于 4DBDh（9:45:58），日期戳域等于 247Ah（1998 年 3 月 26 日）：

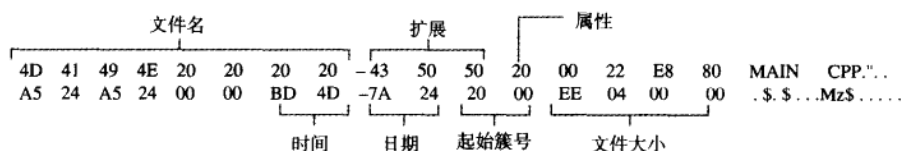


图 14.12 文件目录项示例

在上图中，日期、时间和起始簇号都是 16 位值，以小尾顺序（低字节后面跟高字节）存储。文件大小域是一个双字，也是按照小尾顺序存储的。

14.3.2 MS-Windows 中的长文件名

在 MS-Windows 中，任何长于 11（8+3）个字符的文件名或者大小写混用的文件名都要分配多个目录项。如果目录项的属性值字节等于 0Fh，系统就查看该目录项偏移 0 处的状态字节。如果高 4 位等于 4，则该目录项代表多个长文件名目录项的开始，该字节的低 4 位表示长文件名使用的目录项数目。后面的长文件名目录项中，该域的计数从 $n-1$ 递减至 1（其中 n 代表目录项的总数）。例如，若文件名使用了三个目录项，第一个目录项的状态字节就是 43h，后面目录项的属性字节分别等于 02h 和 01h，如下表所示。

状态字节	含 义
43	表示长文件名共使用三个目录项，该目录项包含文件名的最后一部分
02	该目录项包含文件名的第二部分
01	该目录项包含文件名的第一部分

例子：为了举例说明，我们使用 26 个字符 ABCDEFGHIJKLMNOPQRSTUVWXYZ.TXT 作为文件名在 A 盘根目录下创建一个文本文件，接下来在命令提示符下运行 DEBUG.EXE，把目录扇区装入内存偏移 100h 处，使用 D 命令显示^①：

```
L 100 0 13 5      (装入扇区 13h~17h)
D 100              (在屏幕上显示 100h 偏移处的内容)
```

如图 14.13 所示，Windows 为该文件创建了三个目录项。

^① 参见本书网站上的 DEBUG 教程。



图 14.13 一个长文件名的目录项

我们从 01C0h 处的目录项开始看, 第一个字节是 01, 表示该目录项是长文件名的最后一个目录项, 该目录项包含了长文件名的前 13 个字符 ABCDEFGHIJKLM, 其中每个字符都是 16 位长的 Unicode 字符, 以小尾顺序存储。注意偏移 0B 处的字节为 0F, 表示这是扩展文件名目录项 (MS-DOS 自动忽略任何有该属性值的文件名)。

01A0h 处的目录项包含了长文件名的最后 13 个字符 NOPQRSTUVWXYZ.TXT。

偏移 01E0h 处是系统根据长文件名自动生成的短文件名, 其中包含了长文件名中的前 6 个字符, 其后跟 “~1” 和长文件名中最后的一个句点后的前 3 个字符, 这些字符都是单字节的 ASCII 码字符 (不是 Unicode 字符)。短文件名目录项还包括文件的创建日期和时间、最后访问日期、最后修改日期和时间、起始簇号和文件大小。图 14.14 是 Windows 资源管理器属性对话框显示的信息, 是与原始的目录数据相对应的。

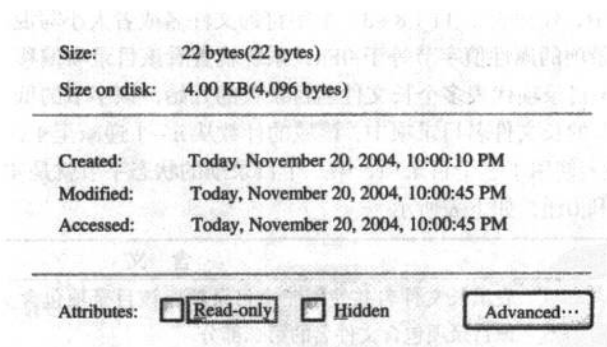


图 14.14 文件属性对话框

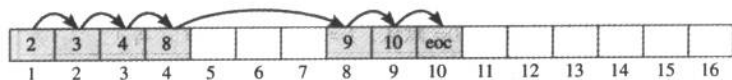
14.3.3 文件分配表 (FAT)

FAT12, FAT16 和 FAT32 文件系统使用一张称为文件分配表 (FAT, File Allocation Table) 的表格记录磁盘上每个文件的位置。FAT 是磁盘上所有簇的映射, 规定了簇和特定文件之间的所有权属关系。文件分配表中的每个项都与特定的簇号相对应, 每个簇包含一个或多个扇区。换句话说, FAT 中的第 10 项标识磁盘上的第 10 个簇, 第 11 项标识磁盘上的第 11 个簇, 依次类推。

在 FAT 中文件以链表标识, 称为簇链。每个 FAT 表项包含标识下一个簇号的整数。File1 和 File2

使用的两个簇链如图 14.15 所示。File1 占用了簇 1, 2, 3, 4, 8, 9 和 10, File2 占用了簇 5, 6, 7, 11 和 12。最后一个 FAT 表项中的 eoc 标记是一个预定义的值, 表示该簇是簇链中的最后一个簇了。

File1: 起始簇号 1, 大小为 7 个簇



File2: 起始簇号 5, 大小为 5 个簇

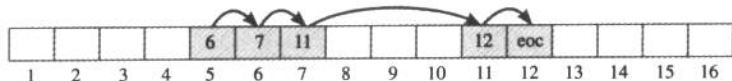


图 14.15 例子: 两个簇链

当创建文件时, 操作系统查找 FAT 中的第一个可用簇。因为常常没有足够的连续簇容纳整个文件, 因此存储文件的簇会产生间隙。上图中, File1 和 File2 都发生了这种情况。当文件修改后存回磁盘, 簇链常常变得更加碎片化。如果很多文件都碎片化了, 整个磁盘的性能就会开始下降, 因为读写磁头必须在不同的磁道之间跳跃以定位文件的全部簇。大多数操作系统都提供了一个内建的磁盘碎片整理工具。

14.3.4 本节习题

1. (对/错) 完整的文件名包括文件的路径和文件的名字。
2. (对/错) 磁盘上文件根的列表称为基目录。
3. (对/错) 文件目录项中包含文件的起始扇区号。
4. (对/错) 目录项的 MS-DOS 日期戳域必须与 1980 相加。
5. 一个 MS-DOS 目录项使用多少字节?
6. 说出 MS-DOS 目录项的 7 个基本域的名字 (不包括保留域)。
7. 在 MS-DOS 文件名项中, 6 个可能的状态字节值是哪些?
8. 说明 MS-DOS 目录项中时间戳域的格式。
9. 当长文件名存储在卷目录中时 (在 MS-Windows 下), 如何识别第一个长文件名目录项?
10. 如果文件名有 18 个字符, 需要几个长文件名目录项?
11. MS-Windows 在原来 MS-DOS 文件目录项的基础上添加了两个新的日期戳域, 它们的名称分别是什么?
12. 挑战: 图解说明顺序使用第 2, 3, 7, 6, 4, 8 簇的文件在文件分配表中的簇链结构。

14.4 读写磁盘扇区 (7305h)

INT 21h 的功能调用 7305h (绝对磁盘读写) 允许读写磁盘的逻辑扇区。像所有 INT 指令一样, 它也是设计运行于 16 位实地址模式下的。我们不会试图在保护模式下调用 INT 21h (或任何其他中断), 因为那将涉及更复杂的问题。

功能 7305h 能够在 Windows 95/98/Me 下的 FAT12, FAT16 和 FAT32 文件系统下工作, 但不能在 Windows NT/2000/XP 下工作, 这是因为 Windows NT/2000/XP 有严格的安全机制, 如果允许任何程序读写磁盘扇区, 那么程序就能轻易绕过文件和目录的共享许可检查。调用 7305h 时需

要传递下面的参数。

AX	7305h
DS:BX	DISKIO 结构变量的段-偏移
CX	0FFFFh
DL	驱动器号 (0=默认, 1=A, 2=B, 3=C, 等)
SI	读写标志

DISKIO 结构包含起始扇区号、要读写的扇区数和用于存放扇区数据的缓冲区的段-偏移地址:

```
DISKIO STRUCT
    startSector DWORD 0           ; 起始扇区号
    numSectors  WORD 1           ; 扇区数
    bufferOfs   WORD OFFSET buffer ; 缓冲区的偏移地址
    bufferSeg   WORD SEG buffer   ; 缓冲区所在的段
DISKIO ENDS
```

下面的例子定义了用于存放扇区数据的输入缓冲区以及一个 DISKIO 结构变量:

```
.data
buffer BYTE 512 DUP(?)
diskStruct DISKIO <>
diskStruct2 DiskIO <10,5> ; 扇区 10,11,12,13,14
```

调用功能 7305h 的时候, SI 中传递的参数决定了是读还是写扇区, 要读扇区则清除位 0, 要写扇区则设置位 0。除此之外, 写扇区时 SI 中的参数的位 13, 14 和 15 要按下表设置。

位 13~15	扇区类型
000	其他/未知
001	FAT 数据
010	目录数据
011	普通文件数据

其余位 (位 1~12) 必须清零。

例子 1: 下面的语句读取驱动器 C 中的一个或多个扇区:

```
mov ax,7305h           ; 绝对读/写
mov cx,0FFFFh          ; 总是这个值
mov dl,3               ; 驱动器 C
mov bx,OFFSET diskStruct ; DISKIO 结构
mov si,0               ; 读扇区
int 21h
```

例子 2: 下面的语句写驱动器 A 的一个或多个扇区:

```
mov ax,7305h           ; 绝对读/写
mov cx,0FFFFh          ; 总是这个值
mov dl,1               ; 驱动器 A
mov bx,OFFSET diskStruct ; DISKIO 结构
mov si,6001h           ; 写普通扇区
int 21h
```

14.4.1 扇区显示程序

下面利用所学的关于扇区的知识, 写一个以 ASCII 码格式显示磁盘某个扇区的程序。程序的伪码如下所示:

询问起始扇区号和驱动器号

```
do while( 击键 <> ESC )
    显示标题
    读取一个扇区
    如果发生了 MS-DOS 错误则退出
    显示一个扇区
    等待击键
    扇区号递增
end do
```

程序清单：以下是 Sector.asm 程序的完整清单，程序可在 Windows 95/98/Me 的实地址模式下运行，但由于 Windows NT/2000/XP 下对磁盘访问有更加严格的安全限制，因此程序不能在这些系统下运行。

```
TITLE Sector Display Program                (Sector.asm)
; Demonstrates INT 21h function 7305h (ABSDiskReadWrite)
; This Real-mode program reads and displays disk sectors.
; Works on FAT16 & FAT32 file systems running under Windows
; 95, 98, and Millenium.

INCLUDE Irvine16.inc

Setcursor PROTO, row:BYTE, col:BYTE
EOLN EQU <0dh,0ah>
ESC_KEY = 1Bh
DATA_ROW = 5
DATA_COL = 0
SECTOR_SIZE = 512
READ_MODE = 0                                ; 用于功能调用 7305h

DiskIO STRUCT
    startSector DWORD ?                    ; 起始扇区号
    numSectors  WORD 1                    ; 扇区数
    bufferOfs   WORD buffer                ; 缓冲区偏移地址
    bufferSeg   WORD, @DATA                ; 缓冲区所在的段
DiskIO ENDS

.data
driveNumber BYTE ?
diskStruct DiskIO <>
buffer BYTE SECTOR_SIZE DUP(0),0          ; 一个扇区

curr_row  BYTE ?
curr_col  BYTE ?

; String resources
strLine    BYTE EOLN,79 DUP(0C4h),EOLN,0
strHeading BYTE "Sector Display Program (Sector.exe)"
           BYTE EOLN,EOLN,0
strAskSector BYTE "Enter starting sector number: ",0
strAskDrive  BYTE "Enter drive number (1=A, 2=B, "
           BYTE "3=C, 4=D, 5=E, 6=F): ",0
strCannotRead BYTE EOLN,"*** Cannot read the sector. "
           BYTE "Press any key...", EOLN, 0
strReadingSector \
           BYTE "Press Esc to quit, or any key to continue..."
           BYTE EOLN,EOLN,"Reading sector: ",0

.code
main PROC
    mov     ax,@data
```



```

    mov     ds,ax
    call    Clrscr
    mov     dx,OFFSET strHeading           ; 显示欢迎信息
    call    Writestring                     ; 询问用户...
    call    AskForSectorNumber

L1:  call    Clrscr
    call    ReadSector                     ; 读取一个扇区
    jc      L2                             ; 如果发生错误则退出
    call    DisplaySector
    call    ReadChar
    cmp     al,ESC_KEY                     ; 按下了 Esc?
    je      L3                             ; 是: 退出
    inc     diskStruct.startSector         ; 下一个扇区
    jmp     L1                             ; 重复循环

L2:  mov     dx,OFFSET strCannotRead       ; 错误消息
    call    Writestring
    call    ReadChar

L3:  call    Clrscr
    exit
main ENDP
;-----
AskForSectorNumber PROC
;
; Prompts the user for the starting sector number
; and drive number. Initializes the startSector
; field of the DiskIO structure, as well as the
; driveNumber variable.
;-----
    pusha
    mov     dx,OFFSET strAskSector
    call    WriteString
    call    ReadInt
    mov     diskStruct.startSector,eax
    call    Crlf
    mov     dx,OFFSET strAskDrive
    call    WriteString
    call    ReadInt
    mov     driveNumber,al
    call    Crlf
    popa
    ret
AskForSectorNumber ENDP
;-----
ReadSector PROC
;
; Reads a sector into the input buffer.
; Receives: DL = Drive number
; Requires: DiskIO structure must be initialized.
; Returns:  If CF=0, the operation was successful;
;           otherwise, CF=1 and AX contains an
;           error code.
;-----
    pusha
    mov     ax,7305h                       ; 绝对磁盘读写
    mov     cx,-1                           ; 总是-1
    mov     bx,OFFSET diskStruct           ; 扇区号
    mov     si,READ_MODE                   ; 读取模式
    int     21h                             ; 读取磁盘扇区
    popa
    ret
ReadSector ENDP

```

```

;-----
DisplaySector PROC
;
; Display the sector data in <buffer>, using INT 10h
; BIOS function calls. This avoids filtering of ASCII
; control codes.
; Receives: nothing. Returns: nothing.
; Requires: buffer must contain sector data.
;-----
    mov     dx,OFFSET strHeading           ; 显示标题
    call    WriteString
    mov     eax,diskStruct.startSector     ; 显示扇区号
    call    WriteDec
    mov     dx,OFFSET strLine             ; 水平直线
    call    WriteString
    mov     si,OFFSET buffer              ; 指向缓冲区
    mov     curr_row,DATA_ROW              ; 设置行、列
    mov     curr_col,DATA_COL
    INVOKE  SetCursor,curr_row,curr_col

    mov     cx,SECTOR_SIZE                ; 循环计数器
    mov     bh,0                          ; 视频页 0
L1:  push    cx                            ; 保存循环计数器
    mov     ah,0Ah                        ; 显示字符
    mov     al,[si]                       ; 从缓冲区中取一个字节
    mov     cx,1                          ; 显示之
    int     10h
    call    MoveCursor
    inc     si                             ; 指向下一个字节
    pop     cx                             ; 恢复循环计数器
    loop   L1                             ; 重复循环
    ret
DisplaySector ENDP

;-----
MoveCursor PROC
;
; Advance the cursor to the next column,
; check for possible wraparound on screen.
;-----
    cmp     curr_col,79                   ; 最后一列?
    jae     L1                            ; 是: 转下一行
    inc     curr_col                      ; 否: 列增 1
    jmp     L2
L1:  mov     curr_col,0                   ; 下一行
    inc     curr_row
L2:  INVOKE  Setcursor,curr_row,curr_col
    ret
MoveCursor ENDP

;-----
Setcursor PROC USES dx,
    row:BYTE, col:BYTE
;
; Set the screen cursor position
;-----
    mov     dh, row
    mov     dl, col
    call    Gotoxy
    ret
Setcursor ENDP
END main

```

程序的核心是 ReadSector 过程，该过程使用 INT 21h 的功能 7305h 从磁盘上读取每个扇区，把读出的扇区数据存放在缓冲区中，然后使用 DisplaySector 过程显示缓冲区数据。

使用 INT 10h：大多数扇区包含的是二进制数据，如果用 INT 21h 显示，ASCII 控制字符将被过滤掉，制表符和换行符将使显示变得不连续。较好的方法是使用 INT 10h 的功能 9，它将 ASCII 码 0~31 显示为图形字符。INT 10h 在 15.4 节描述。由于 INT 10h 的功能 9 并不自动跟随光标，因此要再写一些附加代码在显示完每个字符后把光标向右移动一个字符。SetCursor 过程简化了对 Irvine16 库中的 Gotoxy 过程的调用。

变例：在扇区显示程序的基础上可以创建许多有趣的变例，例如可提示用户输入要显示的扇区范围，每个扇区也可以用十六进制格式显示，还可以允许用户使用 PageUp 和 PageDown 键在扇区之间前后切换，其中的一些功能增强出现在本章的习题中。

14.4.2 本节习题

1. （对/错）在 Windows Me 下可以使用 INT 21h 的功能 7305h 读取硬盘扇区，但是在 Windows XP 下则不可以。
2. （对/错）INT 21h 的功能 7305h 只能在保护模式下读取一个或多个磁盘扇区。
3. INT 21h 的功能 7305h 要求哪些输入参数？
4. 在扇区显示程序中（参见 14.4.1 节），为什么使用 INT 10h 来显示字符？
5. 挑战：在扇区显示程序中（参见 14.4.1 节），如果起始扇区号超出范围会发生什么？

14.5 系统级文件功能调用

在实地址模式下，INT 21h 提供了一些系统服务（如表 14.7 所示），如创建和改变目录、修改文件属性和查找匹配文件等。调用这些服务的时候，功能号要放在 AH 或 AX 中，可能还要用到其他的寄存器来传递输入参数值。下面将详细讲述一些常用的功能，MS-DOS 中断及其描述的详细列表请参见附录 C。

Windows 95/98/Me 支持所有的 MS-DOS INT 21h 功能调用并提供了一些扩展，如允许基于 MS-DOS 的应用程序利用长文件名和卷锁定等特性。INT 21h 的功能调用 7303h（获取剩余磁盘空间）就是一个增强的系统功能调用的例子，该功能可以识别 MS-DOS 不支持的大硬盘。

表 14.7 部分 INT 21h 磁盘服务

功 能 号	说 明
0Eh	设置默认驱动器
19h	获取默认驱动器
7303h	获取磁盘空闲空间
39h	创建子目录
3Ah	删除子目录
3Bh	设置当前目录
41h	删除文件
43h	获取/设置文件属性
47h	获取当前目录路径
4Eh	查找第一个匹配文件

(续表)

功 能 号	说 明
4Fh	查找下一个匹配文件
56h	重命名文件
57h	获取/设置日期和时间
59h	获取扩展的错误信息

14.5.1 获取磁盘剩余空间 (7303h)

INT 21H 功能 7303h 可查询 FAT16 和 FAT32 磁盘的大小以及可用的剩余磁盘空间, 相关信息在一个标准结构 ExtGetDskFreSpcStruc 中返回, 该结构如下所示:

```
ExtGetDskFreSpcStruc STRUC
    StructSize      WORD  ?
    Level           WORD  ?
    SectorsPerCluster  DWORD ?
    BytesPerSector   DWORD ?
    AvailableClusters  DWORD ?
    TotalClusters     DWORD ?
    AvailablePhysSectors  DWORD ?
    TotalPhysSectors  DWORD ?
    AvailableAllocationUnits  DWORD ?
    TotalAllocationUnits  DWORD ?
    Rsvd            DWORD 2 DUP (?)
ExtGetDskFreSpcStruc ENDS
```

(该结构的定义可以在 Irvine16.inc 文件中找到。)下面是结构中每个字段的简要说明。

- StructSize: 返回按字节计算的 ExtGetDskFreSpcStruc 结构的大小, INT 21h 功能 7303h (Get_ExtFreeSpace) 执行的时候, 把结构的大小放在这个成员中。
- Level: 一个输入和返回的层次值, 该值必须初始化为 0。
- SectorsPerCluster: 每簇扇区数。
- BytesPerSector: 每扇区字节数。
- AvailableClusters: 可用簇数。
- TotalClusters: 卷上所有簇的总数。
- AvailablePhysSectors: 卷上可用的未调整或压缩的物理扇区数。
- TotalPhysSectors: 卷上所有的未调整或压缩的物理扇区数。
- AvailableAllocationUnits: 卷上可用的未调整或压缩的分配单元数。
- TotalAllocationUnits: 卷上所有的未调整或压缩的分配单元数。
- Rsvd: 保留成员。

调用该功能: 调用 INT 21h 功能 7303h 的时候, 要求输入以下参数:

- AX 必须等于 7303h。
- ES:DI 必须指向 ExtGetDskFreSpcStruc 结构变量。
- CX 必须包含 ExtGetDskFreSpcStruc 结构变量的大小。
- DS:DX 必须指向包含驱动器名的空字符结尾的字符串。可使用类似于“C:\”的 MS-DOS 类型的驱动器标识符或使用通用卷名标识, 如“\\Server\Share”。

如果功能调用成功执行, 进位标志清零, 并且该结构被填充, 否则进位标志置位。在调用该

功能后, 下面这些类型的计算可能是有用的:

- 要想知道卷有多大 (以千字节为单位), 可以使用下面的公式: $(\text{TotalClusters} * \text{SectorsPerCluster} * \text{BytesPerSector}) / 1024$ 。
- 要想知道卷里面还有多少剩余空间 (以千字节为单位), 公式应该是: $(\text{AvailableClusters} * \text{SectorsPerCluster} * \text{BytesPerSector}) / 1024$ 。

获取磁盘剩余空间的程序

下面的程序使用 INT 21h 功能 7303h 获取 FAT 类型的磁盘分区上的剩余空间信息, 程序既显示了分区的大小也显示了剩余空间的大小, 程序可在 Windows 95/98/Me 下运行, 但不能在 Windows NT/2000/XP 下运行:

```

TITLE Disk Free Space                                (DiskSpc.asm)
INCLUDE Irvine16.inc

.data
buffer ExtGetDskFreSpcStruc <>
driveName BYTE "C:",0
str1 BYTE "Volume size (KB): ",0
str2 BYTE "Free space (KB): ",0
str3 BYTE "Function call failed.",0dh,0ah,0

.code
main PROC
    mov     ax,@data
    mov     ds,ax
    mov     es,ax

    mov     buffer.Level,0                ; 必须为 0
    mov     di,OFFSET buffer              ; ES:DI 指向缓冲区
    mov     cx,SIZEOF buffer              ; 缓冲区的大小
    mov     dx,OFFSET DriveName            ; 指向驱动器名
    mov     ax,7303h                       ; 获取磁盘的剩余空间
    int     21h
    jc      error                          ; 如果 CF=1 则失败

    mov     dx,OFFSET str1                 ; 卷的大小
    call    WriteString
    call    CalcVolumeSize
    call    WriteDec
    call    Crlf

    mov     dx,OFFSET str2                 ; 剩余空间
    call    WriteString
    call    CalcVolumeFree
    call    WriteDec
    call    Crlf
    jmp     quit

error:
    mov     dx,OFFSET str3
    call    WriteString

quit:
    exit
main ENDP

;-----
CalcVolumeSize PROC
;

```

```

; Calculate and return the disk volume size, in kilobytes.
; Receives: buffer variable, a ExtGetDskFreSpcStruc structure
; Returns:  EAX = volume size
; Remarks:  (SectorsPerCluster * 512 * TotalClusters) / 1024
;-----
    mov     eax,buffer.SectorsPerCluster
    shl     eax,9                ; 乘以 512
    mul     buffer.TotalClusters
    mov     ebx,1024
    div     ebx                  ; 返回 KB 值
    ret
CalcVolumeSize ENDP

;-----
CalcVolumeFree PROC
;
; Calculate and return the number of available kilobytes
; on the given volume.
; Receives: buffer variable, a ExtGetDskFreSpcStruc structure
; Returns:  EAX = available space, in kilobytes
; Remarks:  (SectorsPerCluster * 512 * AvailableClusters) / 1024
;-----
    mov     eax,buffer.SectorsPerCluster
    shl     eax,9                ; 乘以 512
    mul     buffer.AvailableClusters
    mov     ebx,1024
    div     ebx                  ; 返回 KB 值
    ret
CalcVolumeFree ENDP
END main

```

14.5.2 创建子目录 (39h)

INT 21h 的功能 39h 创建一个新的子目录, 调用时 DS:DX 中接收一个指针, 该指针指向一个以空字符结尾的包含路径名的字符串。下面的例子演示了如何在默认驱动器根目录下创建一个名为 ASM 的子目录:

```

.data
pathname BYTE "\ASM",0
.code
    mov ah,39h                ; 创建子目录
    mov dx,OFFSET pathname
    int 21h
    jc display_error

```

如果功能调用失败, 则进位标志置位。可能返回的错误码是 3 和 5, 错误 3 (路径未找到) 表示路径名的某部分不存在。假设要求 MS-DOS 创建目录 ASM\PROGNEW, 但是 ASM\PROG 不存在, 系统就会返回错误值 3。错误 5 (拒绝访问) 表示欲创建的目录已经存在或路径中的第一个目录是根目录并且根目录已满。

14.5.3 删除子目录 (3Ah)

INT 21h 功能 3Ah 删除一个目录, 调用时 DS:DX 接收一个指针, 指向包含目标驱动器和路径的以空字符结尾的字符串。如果路径中没有驱动器名, 则使用默认驱动器, 下面的代码删除驱动器 C 下的 \ASM 目录:

```
.data
pathname BYTE 'C:\ASM',0
.code
mov ah,3Ah ; 删除子目录
mov dx,OFFSET pathname
int 21h
jc display_error
```

如果功能调用失败,则进位标志置位。可能返回的错误码是 3 (路径未找到), 5 (访问无效, 目录中还包含文件), 6 (无效句柄) 或者 16 (试图删除当前目录)。

14.5.4 设置当前目录 (3Bh)

INT 21h 的功能 3Bh 设置当前目录。DS:DX 接收一个指针,指向包含目标驱动器和路径的以空字符结尾的字符串。下面的语句将 C:\ASM\PROGS 设置为当前目录:

```
.data
pathname BYTE "C:\ASM\PROGS",0
.code
mov ah,3Bh ; 设置当前目录
mov dx,OFFSET pathname
int 21h
jc display_error
```

14.5.5 获取当前目录 (47h)

INT 21h 的功能 47h 返回一个包含当前目录名的字符串。调用时 DL 中接收驱动器号 (0 = 默认, 1=A, 2=B, 等), DS:SI 指向一个 64 字节的缓冲区, MS-DOS 在缓冲区中存放当前目录的全路径名的以空字符结尾的字符串 (驱动器符和开始的反斜线省略)。在功能返回时如果设置了进位标志,那么 AX 中返回的唯一可能的错误码是 0Fh (无效驱动器号)。

在下面的例子中, MS-DOS 返回默认驱动器上的当前目录路径, 假设当前目录是 C:\ASM\PROGS, 则 MS-DOS 返回的字符串是 ASM\PROGS:

```
.data
pathname BYTE 64 dup(0) ; 将由 MS-DOS 填充的路径
.code
mov ah,47h ; 获取当前的目录路径
mov dl,0 ; 在默认驱动器上
mov si,OFFSET pathname
int 21h
jc display_error
```

14.5.6 获取/设置文件属性 (7143h)

INT 21h 的功能 7143h 能够获取/设置文件的属性以及完成一些其他任务 (在 Windows 9x 中, 它替代了 MS-DOS 的 INT 21h 的功能 39)。调用该功能时通过 DX 传递文件名的偏移地址。要设置文件属性, 把 BL 设为 1 并把 CX 设为表 14.8 中所列出的一个或多个属性, 其中属性值 _A_NORMAL 必须单独使用, 其他属性值可以使用 “+” 操作符联合使用。

表 14.8 文件的属性值 (在 Irvine16.inc 中定义)

值	含 义
_A_NORMAL(0000h)	文件可读可写, 只有单独使用时才有意义
_A_RDONLY(0001h)	文件只读, 不可写
_A_HIDDEN(0002h)	文件是隐藏的, 在普通的文件列表中不出现

(续表)

值	含 义
_A_SYSTEM(0004h)	操作系统的一部分或由操作系统独占使用
_A_ARCH(0020h)	存档文件。应用程序使用该值标记文件的备份或删除

下面的代码把一个文件的属性设置为只读并且隐藏：

```
mov ax,7143h
mov bl,1
mov cx,_A_HIDDEN + _A_RDONLY
mov dx,OFFSET filename
int 21h
```

要获取文件的当前属性值，把 BX 设为 0 再调用该功能，属性值通过 CX 返回，可使用 TEST 指令测试单个属性位是否置位：

```
test cx,_A_RDONLY
jnz readOnlyFile           ; 文件是只读的
```

_A_ARCH 可同其他属性一起使用。

14.5.7 本节习题

1. 使用 INT 21h 的哪个功能可以获得驱动器的簇大小？
2. 使用 INT 21h 的哪个功能可以查询驱动器 C 上还有多少簇是空闲的？
3. 调用 INT 21h 的哪个功能可以创建目录 D:\apps 并使其成为当前目录？
4. 调用 INT 21h 的哪个功能可以使文件只读？

14.6 本章小结

在操作系统层次，知道磁盘确切的几何形状（物理结构）或特定型号的磁盘信息并没有太大的用处。在这种情况下 BIOS 相当于磁盘控制器固件，在磁盘硬件和操作系统之间充当着经纪人的角色。

磁盘的表面被格式化成肉眼不可见的同心圆，称为磁道，数据在磁道上以磁记录的形式存储。平均寻道时间是一种磁盘速度测量指标，另外一些磁盘测速指标包括每分钟转数（r/min）以及数据传输速率（每秒传入/传出的数据量）。

柱面是指读写磁头在某位置时可以访问的所有磁道。随着时间的推移，文件在磁盘上的分布越来越广，也开始变得碎片化，此时文件不再存储在相邻的柱面上。

扇区是磁道上 512 字节长的一部分。物理扇区是制造商采用低级格式化在磁盘上做的磁性标记，它是不可见的。

磁盘的物理参数描述了磁盘结构，使之能被系统 BIOS 读取。一个物理硬盘被分成一个或多个称为分区或卷的逻辑单位。一个硬盘可以有多个分区。扩展分区可被分成无限多个逻辑分区。每个逻辑分区都以一个单独的驱动器符的形式出现，并且可以有和其他分区不同的文件系统。每个主分区都可以包含一个可引导的操作系统。

主引导记录（MBR）是在磁盘上创建第一个分区的时候创建的，位于磁盘的第一个逻辑扇区中，它包含以下内容：

- 磁盘分区表，描述了磁盘上所有分区的大小和位置。
- 一段小程序，用于定位分区引导记录并将控制权转移给它，最后分区引导记录中的代码负

责加载操作系统。

文件系统记录了每个磁盘文件的位置、大小和属性。文件系统提供了逻辑扇区到簇以及簇到文件和目录名的映射,簇是所有文件和目录的基本存储单位。

簇是文件使用空间的最小单位,它包含一个或多个磁盘扇区。文件分配表(FAT)使用簇链记录文件使用的所有簇。

IA-32 系统使用下列文件系统:

- FAT12 文件系统最先在 IBM-PC 磁盘上使用。
- FAT16 文件系统是 MS-DOS 下格式化硬盘时唯一可用的文件系统。
- FAT32 文件系统是随着 Windows 95 的 OEM2 版本的发布而引入的,并且在 Windows 98 下做了进一步的改进。
- NTFS 文件系统仅仅被 Windows NT/2000/XP 所支持。

每个磁盘(FAT 类型或 NTFS 文件系统)都有一个根目录,根目录是磁盘上文件的根列表。根目录中还可以包含其他的目录名,它们称为子目录。

MS-DOS 和 Windows 使用一张称为文件分配表的表格记录磁盘上每个文件的位置和大小,文件分配表是磁盘上所有簇的映射,规定了簇和特定文件之间的所有权属关系。文件分配表中的每个项都与特定的簇号相对应,每个簇包含一个或多个扇区。

在实地址模式下,INT 21h 提供了一些系统服务(如表 14.7 所示),如创建和改变目录、修改文件属性和查找匹配文件等。这些功能在高级语言中不是那么容易获得的。

扇区显示程序读取和显示指定磁盘卷上的特定扇区的信息。

磁盘剩余空间获取程序既显示了特定磁盘卷的大小也显示了剩余空间的大小。

14.7 编程练习

下面的(大部分)练习必须在实地址模式下编译和运行,这里的许多程序都会改变磁盘或目录,因此一定要备份受这些程序影响的任何磁盘,或创建一个临时盘来测试这些程序。无论何时,除非已经仔细调试了这些程序,否则不应该在固定硬盘上运行这些程序。

1. 设置默认驱动器

写一个过程,提示用户输入一个驱动器符(A, B, C 或 D),然后根据用户的输入设置默认驱动器。

2. 磁盘空间

写一个名为 Get_DiskSize 的过程,返回指定驱动器上数据空间的总量。输入:AL = 驱动器号(0=A, 1=B, 2=C, ...),输出:DX:AX = 以字节为单位的数据空间总量。

3. 磁盘剩余空间

写一个过程 Get_DiskFreespace,返回特定驱动器上的剩余空间总量。输入:DS:DX 指向一个包含驱动器符的字符串,输出:EDX:EAX = 以字节为单位的磁盘剩余空间。写一个程序,测试该过程并以十六进制格式显示返回的 64 位结果。

4. 显示文件属性

写一个过程 ShowFileAttributes,通过 DX 接收文件名的偏移地址,该过程在控制台窗口中显示文件的属性,要显示的属性值包括普通(normal)、隐藏(hidden)、只读(read-only)和系统(system)

文件属性。提示：使用 INT 21h 的功能调用 7143h。

编写一个程序调用过程 ShowFileAttributes，向该过程传递一个文件的名字。在运行程序之前，在 Windows 资源管理器中的相应文件名上单击右键，选择属性对话框，选中隐藏和只读属性。此外还可以在 Windows 的命令行上运行 Attrib 命令设置文件的属性值。运行程序并验证显示是否正确。输出示例：

```
temp.txt attributes: Hidden Read-only
```

5. 以簇为单位的磁盘剩余空间

修改 14.5.1 节的磁盘剩余空间程序，使其显示下列信息：

Drive specification:	"C:\"
Bytes per sector:	512
Sectors per cluster:	8
Total Number of clusters:	999999
Number of available clusters:	99999

6. 显示扇区号

以扇区显示程序（参见 14.4.1 节）作为起点，在屏幕顶部显示一个字符串，指明当前的扇区号（以十六进制）以及其所在的驱动器盘符。

7. 十六进制扇区显示

以扇区显示程序（参见 14.4.1 节）作为起点，增加一些代码，允许用户按 F2 键以十六进制显示当前扇区，格式是每行 24 字节。每行的开头应显示该行第一个字节的偏移，共应显示 22 行，最后一行的一部分为空。下面是前两行的输出示例：

```
0000 17311625 25425B75 279A4909 200D0655 D7303825 4B6F9234
0018 273A4655 25324B55 273A4959 293D4655 A732298C FF2323DB
```

第 15 章 BIOS 程序设计

本章要点

- 简介
- INT 16h 键盘中断
- INT 10h 视频程序设计
- 使用 INT 10h 绘图
- 内存映射图形
- 鼠标程序设计

15.1 简介

阅读本章犹如重温历史。在第一台 IBM-PC 出现的时候,大批程序员(包括作者本人)都在盘算着如何深入这个小机箱盒子并直接去控制计算机硬件,Peter Norton 迅速发现了很多有用的秘密信息,这导致了他划时代的 *Inside the IBM-PC* 一书的诞生。IBM 也慷慨地公布了 IBM PC/XT BIOS 的所有汇编语言源代码(作者仍保留着一份),重要的游戏设计者如 Michael Abrash (Quake 和 Doom 的作者)利用他们对于 PC 硬件的了解学会了如何优化视频和声音软件^①。现在,读者也可以加入这备受尊敬的一族,在 DOS 和 Windows 之下,也就是在 BIOS (基本输入输出系统, Basic Input-Output System) 层编写程序。这些知识过时了吗?如果读者正在编写嵌入式应用程序或者想了解计算机 BIOS 是如何设计的,那么答案就是:绝对没有!

本章中的所有程序都是 16 位的实地址模式程序,可以在任何版本的 Windows 下运行本章中给出的程序。通过本章,能够学到以下有用的知识:

- 键盘按键按下的时候发生了什么?
- 如何检查键盘缓冲区,查看是否仍有字符在等待以及如何清除原来的击键码。
- 如何读取非 ASCII 键盘按键,如功能键和光标。
- 如何显示彩色文本,为什么色彩是基于视频显示的 RGB 色彩混合系统的。
- 如何把屏幕分成彩条并且分别滚动。
- 如何绘制 256 色的位映射图像。
- 如何探测鼠标移动和鼠标点击。

15.1.1 BIOS 数据区

如表 15.1 所示, BIOS 数据区包含了 ROM BIOS 服务例程使用的系统数据。例如键盘缓冲区(在偏移 001Eh 处)中包含了等待 BIOS 处理的按键的 ASCII 码和扫描码。

① 重要的例子见 Michael Abrash, *The Zen of Code Optimization*, Coriolis Group Books, 1994。

表 15.1 BIOS 数据区, 在段 0040h 内

十六进制偏移	描 述
0000~0007	串口地址, COM1~COM4
0008~000F	并口地址, LPT1~LPT4
0010~0011	安装的硬件列表
0012	初始化标志
0013~0014	内存大小, 以千字节 (KB) 为单位
0015~0016	I/O 通道内存
0017~0018	键盘状态标志
0019	ALT 数字键工作区
001A~001B	键盘缓冲区指针 (头)
001C~001D	键盘缓冲区指针 (尾)
001E~003D	键盘输入缓冲区
003E~0048	磁盘数据区
0049	当前的视频模式
004A~004B	屏幕列的数目
004C~004D	视频缓冲区的长度, 单位是字节
004E~004F	视频缓冲区的起始偏移地址
0050~005F	光标位置, 视频页 1~8
0060	光标结束行
0061	光标起始行
0062	当前显示的视频页号
0063~0064	显示端口的基地址
0065	CRT 模式寄存器
0066	彩色图形适配器寄存器
0067~006B	磁带数据区
006C~0070	时钟数据区

15.2 INT 16h 键盘中断

在 2.5 节, 我们比较了汇编语言程序员可用的各种输入输出层次的不同。在本章中, 读者将有机会调用 (大部分是) 计算机制造商安装的功能函数, 直接在 BIOS 层次编程。BIOS 层正好是硬件层之上的一层, 在该层次控制计算机有着极大的灵活性。

BIOS 调用 INT 16h 处理键盘输入。BIOS INT 16h 中断服务例程不允许重定向, 但读取扩展键时更容易一些, 每个扩展键都会产生一个 8 位的扫描码 (scan code), 扫描码表可在本书前言部分找到。对 IBM 兼容机而言, 每个键的扫描码都是唯一的。所有的按键都产生扫描码, 但我们通常并不关心 ASCII 字符对应的扫描码, 因为标准化的 ASCII 码几乎在所有的计算机上都是相同的。在 MS-Windows 下按下扩展键时, 其 ASCII 码是 00h 或 E0h, 如下表所示。

按 键	ASCII 码
Ins, Del, PgUp, PgDn, Home, End, 光标键 (上、下、左、右)	E0h
功能键 (F1~F12)	00h

15.2.1 键盘是如何工作的

按键之后会发生一系列事件，这些事件从键盘控制芯片开始，直到字符被放入一个称为键盘输入缓冲区（如表 15.1 所示）的数组中后结束。键盘输入缓冲区在任何时刻最多可容纳 15 个击键，因为每次击键产生两个字节的的数据（ASCII 码 + 扫描码）。当用户按下一个键时产生下列事件：

- 键盘控制器芯片向 PC 的键盘输入端口发送一个 8 位的扫描码。
- 输入端口引发一个中断，中断是预定义的信号，用于通知 CPU 一个输入输出设备需要引起注意。CPU 通过执行 INT 9h 服务例程响应键盘中断。
- INT 9h 服务例程从输入端口获取键盘的扫描码并查找按键对应的 ASCII 码。如果按键有对应的 ASCII 码，INT 9h 终端服务例程就把 ASCII 码和扫描码一起插入到键盘缓冲区中。如果按键的扫描码没有对应的 ASCII 码，键盘缓冲区中的 ASCII 码就设为 0。

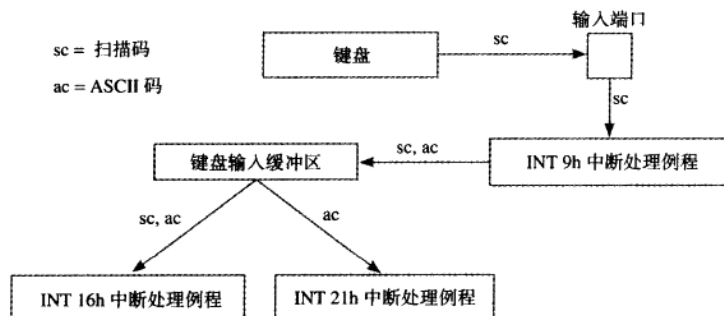


图 15.1 击键的处理顺序

一旦按键的扫描码和 ASCII 码安全地放到了缓冲区中，它们就一直保存在那里，直到被当前运行的程序取出为止。有两种方法取出按键：

- 调用 BIOS INT 16h 的功能，从键盘缓冲区中取得按键的扫描码和 ASCII 码。这在处理没有 ASCII 码的功能键和光标键时非常有用。
- 调用 MS-DOS INT 21h 的功能，从输入缓冲区中取得按键的 ASCII 码。如果按下的是扩展键，必须第二次调用 INT 21h 以返回扫描码。INT 21h 键盘输入在 13.2.3 节中已经解释过了。

15.2.2 INT 16h 功能调用

进行键盘处理时，INT 16h 比 INT 21h 有许多明显的优点。INT 16h 可一次同时返回扫描码和 ASCII 码。另外 INT 16h 还有一些额外的功能，如设置击键重复率和返回键盘状态标志等。击键重复率是指在一直按下某个键时击键的重复速率。在不知道用户按下的是普通键还是扩展键时，INT 16h 通常是可调用的最佳功能。

设置击键重复率(03h)

如下表所示，INT 16h 的功能 03h 允许设置键盘击键重复速率。当一直按着某个键时，在按键开始重复之前通常有 250~1000 ms 的延迟。击键重复速率的取值可以在 1Fh（最慢）到 0（最快）之间。

INT 16h 功能 03h

描述	设置击键重复率	
接收参数	AH = 3	
	AL = 5	
	BH = 重复延迟 (0 = 250 ms; 1 = 500 ms; 2 = 750 ms; 3 = 1000 ms)	
返回值	无	
调用示例	mov ax, 0305h	
	mov bh, 1	; 500 ms 的重复延迟
	mov bl, 0Fh	; 重复速率
	int 16h	

在键盘缓冲区中插入按键 (05h)

如下表所示, INT 16h 的功能 05h 允许在键盘缓冲区中插入按键。一个按键由两个 8 位的整数构成: 一个 ASCII 码和一个键盘扫描码。

INT 16h 功能 05h

描述	在键盘缓冲区中插入按键	
接收参数	AH = 5	
	CH = 扫描码	
	CL = ASCII 码	
返回值	如果键盘缓冲区已满, 则 CF = 1, AL = 1; 否则 CF = 0, AL = 0	
调用示例	mov ah, 5	
	mov ch, 3Bh	; 按键 F1 的扫描码
	mov cl, 0	; ASCII 码
	int 16h	

等待按键 (10h)

INT 16h 的功能 10h 从键盘缓冲区中删除下一个按键。如果缓冲区中没有现成的按键存在, 则键盘处理程序等待用户按下一个键, 如下表所示。

INT 16h 功能 10h

描述	等待键盘按键	
接收参数	AH = 10h	
返回值	AH = 键盘扫描码	
	AL = ASCII 码	
调用示例	mov ah, 10h	
	int 16h	
	mov scanCode, ah	
	mov ASCIICode, al	
注意	如果缓冲区内无按键, 该功能就等待按键。该功能替代了 INT 16h 的功能 00h	

例子程序

下面的键盘显示程序在循环中使用 INT 16h 输入击键并以 ASCII 码和扫描码两种格式显示每个按键, 当按下 Esc 键时程序结束:

```
TITLE Keyboard Display                (Keybd.asm)

; This program displays keyboard scan codes
; and ASCII codes, using INT 16h.
```

```

INCLUDE Irvine16.inc
.code
main PROC
    mov ax,@data
    mov ds,ax
    call ClrScr                ; 清除屏幕

L1: mov ah,10h                ; 键盘输入
    int 16h                   ; 使用 BIOS 功能调用
    call DumpRegs             ; AH=扫描码, AL=ASCII 码
    cmp al,1Bh                ; 按下了 ESC 键?
    jne L1                    ; 否: 重复循环

    call ClrScr                ; 清除屏幕
    exit
main ENDP
END main

```

程序调用 DumpRegs 显示所有的寄存器, 读者只需查看 AH (扫描码) 和 AL (ASCII 码) 的值即可。例如当用户按下 F1 功能键时, 结果显示如下 (3B00h):

EAX=00003B00	EBX=00000000	ECX=000000FF	EDX=00000506
ESI=00000000	EDI=00002000	EBP=0000091E	ESP=00002000
EIP=0000000F	EFL=00003202	CF=0	SF=0 ZF=0 OF=0 AF=0 PF=0

检查键盘缓冲区 (11h)

INT 16h 功能 11h 允许查看键盘缓冲区内是否有按键在等待。如果有按键, 该功能返回按键的 ASCII 码和扫描码。可在执行其他任务的循环内使用该功能调用, 注意该功能调用并不从键盘缓冲区内删除按键。该功能的细节见下表。

INT 16h 功能 11h	
描述	检查键盘缓冲区
接收参数	AH = 11h
返回值	如果有键在等待, 则 ZF = 0, AH = 扫描码, AL = ASCII 码; 否则 ZF = 1
调用示例	<pre> mov ah,11h int 16h jz NoKeyWaiting ; 缓冲区内无按键 mov scanCode,ah mov ASCIICode,al </pre>
注意	并不从键盘缓冲区内删除字符 (如果有的话)

获取键盘标志 (12h)

INT 16h 的功能 12h 返回当前键盘状态标志的相关信息。或许读者已经注意到了字处理程序通常在 CapsLock, NumLock, Insert 等键按下时在屏幕的底端显示标志或记号, 这是通过不断检查键盘标志监视其变化而做到的。

INT 16h 功能 12h	
描述	获取键盘状态标志
接收参数	AH = 12h
返回值	AH = 键盘状态标志的一份副本

(续表)

INT 16h 功能 12h	
调用示例	<pre>mov ah,12h int 16h mov keyFlags,ax</pre>
注意	键盘标志位于 BIOS 数据区中的 00417h~00418h 处

如表 15.2 所示, 键盘标志是非常有趣的, 因为它包含了很多关于用户在对键盘做什么的信息: 用户在按下左 Shift 键还是右 Shift 键? 是否同时还按下了 Alt 键? 这种问题可以通过 INT 16h 来回答, 在一些特殊键 (CapsLock, ScrollLock, NumLock, Insert 等) 按下或处于锁定状态的时候, 对应的键盘标志位置 1。在 Windows 95/98 下, 也可以通过直接读取段 0040h 偏移 17h、18h 处的字节得到。

表 15.2 键盘标志值*

位 号	描 述
0	右 Shift 键按下
1	左 Shift 键按下
2	Ctrl 键按下
3	Alt 键按下
4	ScrollLock 键锁定
5	NumLock 键锁定
6	CapsLock 键锁定
7	Insert 键按下
8	左 Ctrl 键按下
9	左 Alt 键按下
10	右 Ctrl 键按下
11	右 Alt 键按下
12	Scroll 键按下
13	NumLock 键按下
14	CapsLock 键按下
15	SysReq 键按下

*来源: Ray Duncan 所著的 *Advanced MS-DOS Programming*, 第二版, Microsoft Press, 1988, 第 586~587 页。

清空键盘缓冲区

程序中经常包含只能由特定按键才能中断的循环, 例如游戏程序在显示动画时必须检查用户是否按下了方向键或其他特殊键。用户经常会按下大量的无关按键, 以至于键盘缓冲区被填满, 但在用户按下正确的按键时, 我们期望程序能够立即对命令做出反应。

使用 INT 16h 的功能, 可以检查是否有按键在等待 (功能 11h), 也可以从缓冲区中删除按键 (功能 10h)。下面的程序说明了 ClearKeyboard 过程的用法, 该过程在循环中检查特定按键的扫描码时不断清空缓冲区。为便于测试, 程序只检测了 Esc 键, 但实际上它可用来检查任何按键:

```
TITLE Testing ClearKeyboard      (ClearKbd.asm)

; This program shows how to clear the keyboard
; buffer while waiting for a particular key.
```



```

; To test it, rapidly press random keys to fill
; up the buffer. When you press Esc, the program
; program ends immediately.

INCLUDE Irvine16.inc
ClearKeyboard PROTO, scanCode:BYTE
ESC_key = 1 ; 扫描码

.code
main PROC
L1:
    ; Display a dot, to show program's progress
    mov ah,2
    mov dl, '.'
    int 21h
    mov eax,300 ; 延时 300 ms
    call Delay

    INVOKE ClearKeyboard,ESC_key ; 检查 ESC 键
    jnz L1 ; 如果 ZF=0 则继续循环

quit:
    call Clrscr
    exit
main ENDP

;-----
ClearKeyboard PROC,
    scanCode:BYTE
;
; Clears the keyboard while checking for a
; particular scan code.
; Receives: keyboard scan code
; Returns: Zero flag set if the ASCII code is
; found; otherwise, Zero flag is clear.
;-----
    push ax
L1:
    mov ah,11h ; 检查键盘缓冲区
    int 16h ; 按下任何一个键了吗?
    jz noKey ; 否: 退出
    mov ah,10h ; 是: 从缓冲区中删除
    int 16h
    cmp ah,scanCode ; 是否是退出按键?
    je quit ; 是: 现在退出 (ZF=1)
    jmp L1 ; 否: 再次检查缓冲区

noKey:
    ; 无按键
    or al,1 ; 清除零标志
quit:
    pop ax
    ret
ClearKeyboard ENDP
END main

```

该程序每隔 300 ms 在屏幕上显示一个圆点。测试时随机按下的任意键都将被忽略并从键盘缓冲区中删除。一旦按下 Esc 键，程序就会立即终止。

15.2.3 本节习题

1. 哪个中断（16h 或 21h）最适合用于读入包括功能键和其他扩展键的用户输入？
2. 键盘输入的字符在等待应用程序处理时存储在内存中的哪个位置？
3. INT 9h 服务例程执行哪些操作？
4. INT 16h 的哪个功能可以把键盘按键插入键盘缓冲区？
5. INT 16h 的哪个功能可以删除键盘缓冲区中的下一个按键？
6. INT 16h 的哪个功能检查键盘缓冲区并返回第一个输入按键的扫描码和 ASCII 码？
7. INT 16h 功能 11h 是否从键盘缓冲区中删除一个字符？
8. INT 16h 的哪个功能可以返回键盘标志字节的值？
9. 键盘标志字节中的哪个位表示 ScrollLock 键被按下了？
10. 写语句，读入键盘标志字节并重复循环直到 Ctrl 键按下为止。
11. 挑战：15.2.2 节中的 ClearKeyboard 过程只能检查一个键盘扫描码，现在假设程序必须检查多个扫描码（例如 4 个光标键），如果要使之可行，需要对程序做哪些修改？不必写实际代码，给出你的修改建议。

15.3 INT 10h 视频程序设计

15.3.1 背景知识

三种访问方式（层次）

文本模式下应用程序在屏幕上显示字符的时候，可在下列三种输出方式中进行选择：

- 通过 MS-DOS 层访问。任何运行或模拟 MS-DOS 的计算机都能够使用 INT 21h 在视频显示上显示文本，输入输出可以非常容易地重定向到其他设备上，如打印机和磁盘。但这种访问模式输出较慢并且不能控制文本的颜色。
- 通过 BIOS 层访问。使用 INT 10h BIOS 服务输出字符。执行起来比 INT 21h 快得多，而且允许指定文本的颜色。在填充大块屏幕区域时，通常能感觉到轻微的延迟。输出不能重定向。
- 直接视频访问。字符直接送视频 RAM，因此执行是瞬时的。输出不可重定向。在 MS-DOS 时代，字处理程序和电子表格程序都采用了这种方法（在 Windows NT/2000/XP 下这种方法仅限于全屏模式）。

应用程序在选择使用何种访问方式时不尽相同。那些要求高性能的程序选择直接视频访问，其他一些则选择通过 BIOS 层访问。当屏幕输出需要重定向或要和其他程序共享屏幕时，一般通过 MS-DOS 层访问。应该说明的是，MS-DOS 中断调用 BIOS 过程来完成它们的任务，而 BIOS 过程又使用直接视频访问在屏幕上输出结果。

在全屏幕模式下运行程序

使用视频 BIOS 绘制图形的程序应该在下列环境下执行：

- 纯 MS-DOS。

- Linux 下的 DOS 模拟器。
- MS-Windows 下的全屏模式。

在 MS-Windows 下有两种方法切换到全屏模式：

- 在 Windows XP 下为程序的可执行文件创建一个快捷方式。打开快捷方式的 Properties 对话框，选择 Options，然后在 Display Options 组中选择全屏幕模式选项。
- 从开始菜单中打开一个命令行窗口，然后按下 Alt-Enter 键切换到全屏模式。使用 cd (改变目录) 命令进入程序可执行文件所在的目录，输入程序的名字运行程序。Alt-Enter 是一个开关，再次按下它就可以返回到窗口模式。

理解视频文本

(在基于 Intel 的系统上) 视频模式有两种类型：文本模式和图形模式。程序可运行于文本模式或图形模式下，但不能同时运行于这两种模式下：

- 在文本模式下，程序只能在屏幕上显示 ASCII 字符。BIOS 内建的字符生成器为每个字符生成一个位映射图形。在文本模式下，程序不能绘制任意的线条和图形。
- 在图形模式下，程序能够控制每个像素点。MS-DOS 图形模式下没有内建的绘制线条和图形的功能调用，操作只能通过最基本的像素点操作进行。在图形模式下，可以使用内建的功能调用显示文本，还可以替换内建的默认字体。MS-Windows 在图形模式下提供了很多绘制线条和图形的函数。

计算机引导到 MS-DOS 下之后，视频控制器被设置为视频模式 3 (彩色文本，25 行 80 列)。在文本模式下，行从屏幕最顶端 (行 0) 开始计算，行高是当前活跃字体的一个字符单元的高度；列从屏幕的最左边 (列 0) 开始计算，列宽是一个字符单元的宽度。

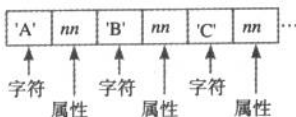
字体：字符是通过驻留在内存的字体表生成的，BIOS 允许程序在运行时重写字体表，以显示自定义的字体。

视频文本页：文本模式下的视频内存被分成多个独立的视频页，每一页都可以容纳整个屏幕的文本。应用程序可以在显示一页的时候写另外的隐藏页，并可以在页之间快速来回切换。在追求高性能的时代，MS-DOS 程序常常要在内存中同时保留若干个文本页。随着当前图形接口的流行，文本页的特性已经不再重要了 (INT 10h 的功能 05h 可设置当前的视频页，本章没有讲述这个功能调用)。默认的视频页是页 0。

属性：如下图所示，在屏幕上显示的每个字符都有一个控制字符颜色 (称为前景色) 和字符之后的屏幕颜色 (称为背景色) 的属性字节。



视频显示的每个位置都存放着一个字符以及其属性 (颜色)。属性以一个单独的字节存储，在内存中的位置是紧跟在字符后面。在下图中，屏幕上的三个位置包含字母 ABC：

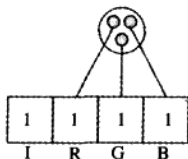


闪烁：视频显示上的字符可以闪烁。视频控制器以预定的频率反转字符的前景色和背景色来做到这一点。当 PC 引导到 MS-DOS 模式下时，默认允许闪烁，也可以使用视频 BIOS 功能关闭闪烁。在 MS-Windows 下打开 MS-DOS 模拟窗口时，默认情况下闪烁是禁止的。

15.3.2 色彩的控制

基色的混合

CRT 视频显示上的每个像素都是通过三种独立的电子束产生的：红、绿、蓝，还有一个通道用来控制总体的亮度或每个像素的亮度，因此所有可用的文本颜色都可以用下表中的 4 位二进制值来表示（I = 亮度、R = 红、G = 绿、B = 蓝）。下图显示了一个白色像素的构成：



通过混合三种基色就可以产生新的色彩（见表 15.3）。进一步地，打开亮度位可以使得混合后的色彩更亮些。

表 15.3 色彩混合的例子

混合的三种基色	得到的颜色	打开亮度位
红 + 绿 + 蓝	浅灰	白
绿 + 蓝	青	浅青
红 + 蓝	洋红	浅洋红
红 + 绿	棕	黄
（无色彩）	黑色	暗灰

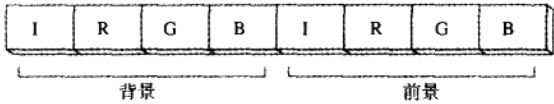
MS-DOS 风格的基色和混合色彩如表 15.4 所示，表中包含了所有可能的 4 位色彩值。右边列中的每种色彩都打开了亮度位。

表 15.4 四位色彩的编码

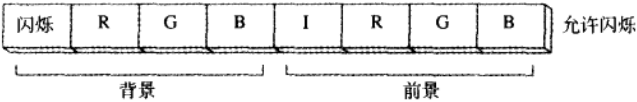
IRGB	色 彩	IRGB	色 彩
0000	黑	1000	灰
0001	蓝	1001	浅蓝
0010	绿	1010	浅绿
0011	青	1011	浅青
0100	红	1100	浅红
0101	洋红	1101	浅洋红
0110	棕	1110	黄
0111	浅灰	1111	白

属性字节

在彩色文本模式下，每个字符都有一个属性字节，该字节由两个 4 位的色彩代码构成，即前景色和背景色：



闪烁：这个简单的色彩方案稍微有点复杂，那就是如果当前的显卡允许闪烁，背景色高位控制着字符是否闪烁。在该位置位时，字符闪烁显示：



在允许闪烁时，表 15.4 中只有左边一栏的低亮度颜色可用做背景色（黑、蓝、绿、青、红、洋红、棕和浅灰）。MS-DOS 引导时默认的配色方案是二进制 00000111（黑色背景，浅灰前景）。

构造属性字节：使用汇编器的 SHL 操作符把背景色值左移 4 位，然后与前景色进行或操作，这样就可以用两种颜色（前景色和背景色）构造一个属性字节了。例如下面的语句创建一个蓝色背景浅灰色文字的的属性字节：

```
blue = 1
lightGray = 111b
mov bh,(blue SHL 4) OR lightGray          ; 00010111
```

下面的语句创建红色背景白色文字的属性值：

```
white = 1111b
red = 100b
mov bh,(red SHL 4) OR white                ; 01001111
```

下面的语句在棕色背景上产生蓝色字符：

```
blue = 1
brown = 110b
mov bh,((brown SHL 4) OR blue)             ; 01100001
```

在不同的操作系统下运行同一程序时字体和颜色会稍微有所不同。例如，在 Windows 2000/XP 下闪烁是被禁止的，除非切换到全屏模式。使用 INT 10h 显示图形时也是如此。

15.3.3 INT 10h 视频功能调用

表 15.5 列出了最常用的 INT 10h 功能调用，后面对每个功能调用都单独进行讨论并给出一些小的示例程序。对功能 0Ch 和 0Dh 的介绍要推迟到本章介绍图形的部分（15.4 节）。

表 15.5 精选的 INT 10h 功能调用

功 能 号	描 述
0	把视频显示设置为文本或图形显示模式
1	设置光标起始和结束线，控制光标的形状和大小
2	设置光标在屏幕上的位置
3	获取光标的屏幕位置和大小
6	上卷当前的视频页，上卷空出的行用空行代替
7	下卷当前的视频页，下卷空出的行用空行代替
8	读取当前光标所在位置的字符及其属性
9	在当前光标位置显示字符及其属性

(续表)

功 能 号	描 述
0Ah	在当前光标位置显示字符，不改变原来的色彩属性
0Ch	图形视频模式下在屏幕上写一个像素点（参见附录 C）
0Dh	读取指定位置像素点的色彩值（参见附录 C）
0Fh	获取视频模式信息
10h	切换闪烁/亮度模式
13h	显示字符串
1Eh	以电传模式向屏幕上显示字符串

在调用 INT 10h 之前保护通用寄存器（使用 PUSH 指令）是个好主意，因为不同版本的 BIOS 保护的寄存器并不相同。

设置视频模式（00h）

INT 10h 功能 0 允许把当前视频模式设置为文本或图形模式，表 15.6 列出了一些可能用到的文本模式的编号。

表 15.6 INT 10h 可以识别的视频文本模式

模 式	分辨率（列 × 行）	颜色数量
0	40 × 25	16
1	40 × 25	16
2	80 × 25	16
3	80 × 25	16
7*	80 × 25	2
14h	132 × 25	16

* 单色监视器

在设置新的视频模式之前，最好首先获取（使用 INT 10h 功能 0Fh）当前的视频模式并保存在变量中，在程序退出时就可以利用该值恢复原始的视频模式了。

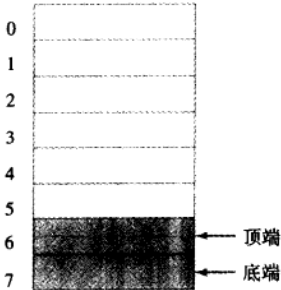
INT 10h 功能 0	
描述	设置视频模式
接收参数	AH = 0 AL = 视频模式
返回值	无
调用示例	mov ah, 0 mov al, 3 ; 视频模式 3（文本模式） int 10h
注意	在调用该功能之前，如果未设置 AL 中的最高位，则屏幕自动清除

设置光标线（01h）

如下表所示，INT 10h 的功能 01h 用来设置文本光标的大小。通过定义起始扫描行和结束扫描线可以控制文本光标的显示大小。应用程序可以通过设置光标的大小以显示当前操作的状态。例如，文本编辑器可能会在锁定 NumLock 键时增大光标，再次按下 NumLock 的时候，光标又恢复成原始大小。

INT 10h 功能 01h	
描述	设置光标起始和结束线
接收参数	AH = 01h CH = 起始线 CL = 结束线
返回值	无
调用示例	<pre>mov ah,1 mov cx,0607h ; 彩色模式下默认的光标大小 int 10h</pre>
注意	彩色显示模式下使用 8 线方式

光标可描述为一系列的水平线，其中第 0 线在最顶端。默认情况下光标从第 6 线开始，到第 7 线结束，如下图所示。



设置光标位置 (02h)

INT 10h 的功能 02h 把光标定位在选定视频页的特定行列位置，如下表所示。

INT 10h 功能 02h	
描述	设置光标位置
接收参数	AH = 2 DH, DL = 行、列值 BH = 视频页
返回值	无
调用示例	<pre>mov ah,2 mov dh,10 ; 行 10 mov dl,20 ; 列 20 mov bh,0 ; 视频页 0 int 10h</pre>
注意	在 80×25 模式下，DH 的取值范围是 0~24，DL 的取值范围是 0~79

获取光标的位置和大小 (03h)

下表所示的 INT 10h 的功能 3 返回光标的行/列位置以及决定光标大小的起始线和结束线。当用户在菜单周围移动鼠标的时候，这个功能是相当有用的。根据光标的位置，就可以知道哪个菜单项被选中了。

INT 10h 功能 03h	
描述	获取光标的位置和大小
接收参数	AH = 3 BH = 视频页

(续表)

INT 10h 功能 03h	
返回值	CH, CL = 光标的起始、结束扫描线 DH, DL = 光标的行、列位置
调用示例	<pre> mov ah,3 mov bh,0 ; 视频页 0 int 10h mov cursor,CX mov position,DX </pre>

显示和隐藏光标：如果能在显示菜单、不断向屏幕输出或读取鼠标输入时暂时隐藏光标，那将是非常有用的。把光标的顶线设置为非法值（较大值）可以隐藏光标，把光标线恢复为默认值（线 6 和线 7）可以重新显示光标：

```

HideCursor PROC
    mov ah,3          ; 获取光标的大小
    int 10h
    or ch,30h         ; 把起始线设为非法值
    mov ah,1          ; 设置光标的大小
    int 10h
    ret
HideCursor ENDP

ShowCursor PROC
    mov ah,3          ; 获取光标的大小
    int 10h
    mov ah,1          ; 设置光标的大小
    mov cx,0607h      ; 默认大小
    int 10h
    ret
ShowCursor ENDP

```

这里忽略了用户在隐藏光标之前把光标值设为非默认值的情况。下面是另一个版本的 ShowCursor 过程，该过程只是简单地把 CH 的高 4 位清除而不改变存储的光标线的低 4 位：

```

ShowCursor PROC
    mov ah,3          ; 获取光标大小
    int 10h
    mov ah,1          ; 设置光标大小
    and ch,0Fh        ; 清除高 4 位
    int 10h
    ret
ShowCursor ENDP

```

遗憾的是，这种隐藏光标的方法并不总是能够工作。另外一种变通的方法是使用 INT 10h 的功能 02h 把光标定位于屏幕边缘（如第 25 行）之外。

上卷窗口（06h）

INT 10h 功能 6 上卷屏幕上矩形区域内（称为窗口）的所有文本，窗口是使用左上角和右下角的行列坐标来定义的。默认的 MS-DOS 屏幕从顶端开始计算有 25 行（0~24），从左边开始计算有 80 列（0~79）。因此覆盖整个屏幕的窗口从坐标(0,0)到坐标(24,79)。在图 15.2 中，CH/CL 寄存器定义左上角的行列位置，DH/DL 定义右下角的行列位置。

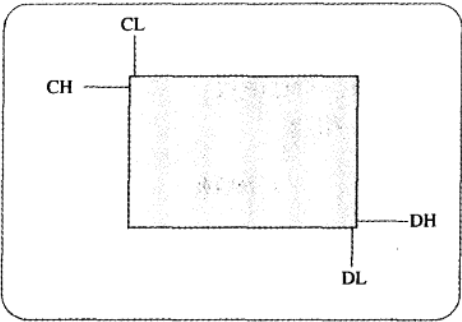


图 15.2 使用 INT 10h 定义一个窗口

窗口上卷时，底端的行由空行代替。如果所有的行都上卷了，那么窗口就被清空了（变为空白），上卷时移出窗口的行不能恢复。下表描述了 INT 10h 的功能 6。

INT 10h 功能 6	
描述	上卷窗口
接收参数	AH = 6 AL = 要上卷的行数 (0 = 全部) BH = 空白区域的视频属性 CH, CL = 窗口左上角的行列位置 DH, DL = 窗口右下角的行列位置
返回值	无
调用示例	mov ah,6 ; 上卷窗口 mov al,0 ; 整个窗口 mov ch,0 ; 左上角的行位置 mov cl,0 ; 左上角的列位置 mov dh,24 ; 右下角的行位置 mov dl,79 ; 右下角的列位置 mov bh,7 ; 空白区域的属性 int 10h ; 调用 BIOS

例子：在窗口内显示文本

在使用 INT 10h 功能 6（或 7）卷动窗口的时候，也同时设置了窗口内被卷动的行的属性值。如果随后使用DOS功能调用在窗口内部显示文本的时候，文本将使用新设置的前景和背景色。下面的程序（TextWin.asm）演示了该技巧：

```
TITLE Color Text Window                                (TextWin.asm)

; Display a color window and write text inside.
INCLUDE Irvine16.inc
.data
message BYTE "Message in Window", 0

.code
main PROC
    mov ax,@data
    mov ds,ax
; Scroll a window.
```

```

mov     ax,0600h                ; 滚动窗口
mov     bh,(blue SHL 4) OR yellow ; 属性
mov     cx,050Ah                ; 左上角
mov     dx,0A30h                ; 右上角
int     10h

; Position the cursor inside the window.
mov     ah,2                    ; 设置光标位置
mov     dx,0714h                ; 行 7 列 20
mov     bh,0                    ; 视频页 0
int     10h

; Write some text in the window.
mov     dx,OFFSET message
call    WriteString

; Wait for a keypress.
mov     ah,10h
int     16h
exit
main ENDP
END main

```

下卷窗口 (07h)

除了窗口内文本的移动方向是向下的以外，下卷窗口与 06h 的功能基本相同，它们的输入参数也是相同的。

读取字符及其属性 (08h)

如下表所示，INT 10h 功能 8 返回当前光标位置处的字符及其属性。程序可以利用该功能调用直接从屏幕上读取文本（称为抓屏技术）。程序可为听力受损的用户把文本转换成语音。

INT 10h 功能 08h	
描述	读取当前光标位置的字符及其属性
接收参数	AH = 8 BH = 视频页
返回值	AL = 字符的 ASCII 码 AH = 字符的属性值
调用示例	<pre> mov ah,8 mov bh,0 ; 视频页 0 int 10h mov char,al ; 保存字符 mov attrib,ah ; 保存属性 </pre>

显示字符并设置其属性 (09h)

INT 10h 功能 9 在当前的光标位置显示彩色字符。正如下表中代码演示的那样，这个功能可以显示任何的 ASCII 字符，包括 ASCII 码为 1~31 的特殊 IBM 图形字符。

INT 10h 功能 09h	
描述	显示字符并设置其属性
接收参数	AH = 9 AL = 字符的 ASCII 码 BH = 视频页 BL = 属性 CX = 重复次数

(续表)

INT 10h 功能 09h

返回值	无
调用示例	<pre> mov ah,9 mov al,'A' ; ASCII 字符 mov bh,0 ; 视频页 0 mov bl,71h ; 属性 (浅灰底色、蓝色前景色) mov cx,1 ; 重复次数 int 10h </pre>
注意	在显示字符之后并不前进光标。在文本和图形模式下均可调用该功能

CX 中的重复次数决定字符重复显示多少次 (重复显示时字符不应超出当前行的结尾)。在显示完字符之后, 如果还要继续显示字符, 必须调用 INT 10h 功能 2 前进光标。

显示字符 (0Ah)

INT 10h 功能 0Ah 在当前光标位置显示字符而不改变当前屏幕位置的字符属性。正如下表所示, 该功能除了不需要指定属性值之外, 其他方面与功能 9 是相同的。

INT 10h 功能 0Ah

描述	显示字符并设置其属性
接收参数	AH = 0Ah AL = 字符 BH = 视频页 CX = 重复次数
返回值	无
调用示例	<pre> mov ah,0Ah mov al,'A' ; ASCII 字符 mov bh,0 ; 视频页 0 mov cx,1 ; 重复次数 int 10h </pre>
注意	在显示字符之后不前进光标

获取视频模式信息 (0Fh)

INT 10h 功能 0Fh 返回关于当前视频模式的信息, 包括模式号、显示的列数以及当前活跃视频页号, 具体如下所示。

INT 10h 功能 0Fh

描述	获取当前视频模式信息
接收参数	AH = 0Fh
返回值	AL = 当前的显示模式 AH = 列数 (字符数或像素数) BH = 活跃的视频页号
调用示例	<pre> mov ah,0Fh int 10h mov vmode,al ; 保存模式 mov columns,ah ; 保存列数 mov page,bh ; 保存页号 </pre>
注意	在文本和视频模式下均可工作

切换闪烁/亮度模式 (10h:03h)

INT 10h 功能 10h 有许多有用的子功能, 子功能 03h 允许把色彩属性值的最高位设置为控制字符的色彩亮度或控制字符是否闪烁, 如下表所示。

INT 10h 功能 10h 子功能 03h	
描述	切换闪烁/亮度模式
接收参数	AH = 10h AL = 3 BL = 闪烁模式 (0 = 允许亮度, 1 = 允许闪烁)
返回值	无
调用示例	<pre> mov ah,10h mov al,3 mov bl,1 ; 允许闪烁 int 10h </pre>
注意	切换使屏幕上文本处于闪烁模式或高亮模式。在 MS-Windows 中必须运行于全屏模式下字符才能闪烁

以电传打字机方式显示字符串 (13h)

下表所示的 INT 10h 功能 13h 从屏幕上指定的位置开始显示字符串。字符串可以选择同时包含字符及其属性值 (参见本书附带代码 ch15 目录下示例程序 colorst2.asm)。

INT 10h 功能 13h	
描述	以电传打字机方式显示字符串
接收参数	AH = 13h AL = 显示模式 (见注意部分) BH = 视频页 BL = 属性值 (如果 AL = 00h 或 01h) CX = 字符串长度 (以字符计数) DH, DL = 屏幕上显示起始位置的行、列值 ES:BP = 字符串的段:偏移地址
返回值	无
调用示例	<pre> .data colorString BYTE 'A',1Fh,'B',1Ch,'C',1Bh,'D',1Ch row BYTE 10 column BYTE 20 .code mov ax,SEG colorString ; 设置 ES 段 mov es,ax mov ah,13h ; 显示字符串 mov al,2 ; 显示模式 mov bh,0 ; 视频页 mov cx,(SIZEOF colorString) / 2 ; 字符串长度 mov dh,row ; 起始行 mov dl,column ; 起始列 mov bp,OFFSET colorString ; 字符串的偏移地址 int 10h </pre>

(续表)

INT 10h 功能 13h

- 注意 在显卡处于文本或图形方式下时均可调用。
- 显示模式的值:
- 00h = 字符串只包含字符码, 在显示之后不更新光标位置, 属性值在 BL 中
 - 01h = 字符串只包含字符码, 在显示之后更新光标位置, 属性值在 BL 中
 - 02h = 字符串包含字符码及其属性值, 在显示之后不更新光标位置
 - 03h = 字符串包含字符码及其属性值, 在显示之后更新光标位置

例子: 显示彩色字符串

下面的程序(ColorStr.asm)在控制台上显示一个字符串, 其中的每个字符都使用不同的颜色。如果想要看到闪烁的字符, 则程序必须运行于全屏模式之下。在默认情况下, 程序是允许闪烁的, 但是可以删除对 EnableBlinking 的调用, 这样就能见到暗灰色背景的字符串了。

```

TITLE Color String Example                                (ColorStr.asm)
INCLUDE Irvine16.inc
.data
ATTRIB_HI = 10000000b
string BYTE "ABCDEFGHIJKLMOP"
color  BYTE (black SHL 4) OR blue
.code
main PROC
    mov     ax,@data
    mov     ds,ax
    call    ClrScr
    call    EnableBlinking          ; 这是可选的
    mov     cx,SIZEOF string
    mov     si,OFFSET string

L1:  push    cx                      ; 保存循环计数器
     mov     ah,9                  ; 显示字符/属性
     mov     al,[si]               ; 要显示的字符
     mov     bh,0                  ; 视频页 0
     mov     bl,color             ; 属性
     or      bl,ATTRIB_HI          ; 设置闪烁/亮度位
     mov     cx,1                  ; 显示一次
     int     10h
     mov     cx,1                  ; 前进光标至
     call    AdvanceCursor         ; 屏幕上的下一列
     inc     color                 ; 下一个颜色
     inc     si                    ; 下一个字符
     pop     cx                    ; 恢复循环计数器
     loop    L1
     call    CrLf
     exit
main ENDP

;-----
EnableBlinking PROC
;
; Enable blinking (using the high bit of color
; attributes). In MS-Windows, this only works if
; the program is running in full screen mode.
; Receives: nothing. Returns: nothing
;-----

```

```

    push    ax
    push    bx
    mov     ax,1003h          ; 切换闪烁/亮度
    mov     bl,1              ; 允许闪烁
    int     10h
    pop     bx
    pop     ax
    ret
EnableBlinking ENDP

AdvanceCursor 过程可用于任何调用 INT 10h 文本函数的程序。
;-----
AdvanceCursor PROC
;
; Advances the cursor n columns to the right.
; (Cursor does not wrap around to the next line.)
; Receives: CX = number of columns
; Returns: nothing
;-----
    pusha
L1:
    push    cx                ; 保存循环计数器
    mov     ah,3              ; 获取光标位置
    mov     bh,0              ; 至 DH、DL 中
    int     10h               ; 改变了 CX 寄存器
    inc     dl                 ; 列增加
    mov     ah,2              ; 设置光标位置
    int     10h
    pop     cx                ; 恢复循环计数器
    loop    L1                ; 下一列
    popa
    ret
AdvanceCursor ENDP
END main

```

15.3.4 库过程中的例子

下面看一下 Irvine16 链接库中的两个很简单但非常有用的过程：Gotoxy 和 Clrscr。

Gotoxy 过程

Gotoxy 过程设置视频页 0 上的光标位置：

```

;-----
Gotoxy PROC
;
; Sets the cursor position on video page 0.
; Receives: DH,DL = row, column
; Returns: nothing
;-----
    pusha
    mov     ah,2              ; 设置光标位置
    mov     bh,0              ; 视频页 0
    int     10h
    popa
    ret
Gotoxy ENDP

```

Clrscr 过程

Clrscr 过程清除屏幕并把光标定位在视频页 0 的位置——行 0 列 0:

```

;-----
Clrscr PROC
;
; Clears the screen (video page 0) and locates the cursor
; at row 0, column 0.
; Receives: nothing
; Returns:  nothing
;-----
    pusha
    mov ax,0600h          ; 上卷窗口
    mov cx,0              ; 左上角 (0,0)
    mov dx,184Fh          ; 右下角 (24,79)
    mov bh,7              ; 普通属性
    int 10h               ; 调用 BIOS
    mov ah,2              ; 把光标定位在 (0,0) 处
    mov bh,0              ; 视频页 0
    mov dx,0              ; 行 0 列 0
    int 10h
    popa
    ret
Clrscr ENDP

```

15.3.5 本节习题

1. 本节开始提到的三种视频显示访问层次分别是什么?
2. 通过哪种层次访问的输出速度最快?
3. 如何在全屏模式下运行程序?
4. 计算机引导到 MS-DOS 下时, 默认的视频模式是什么?
5. 视频显示的每个位置存放着单个字符的哪些信息?
6. 要在视频显示上产生任何颜色, 都需要哪些电子束?
7. 说明视频属性字节中前景色和背景色的位置。
8. INT 10h 的哪个功能在屏幕上定位光标?
9. INT 10h 的哪个功能把矩形窗口内的文本上卷?
10. INT 10h 的哪个功能在当前光标位置以特定属性显示字符?
11. INT 10h 的哪个功能设置光标的大小?
12. INT 10h 的哪个功能获取当前的视频模式?
13. 使用 INT 10h 设置光标位置需要哪些参数?
14. 如何隐藏光标?
15. 上卷窗口内的文本时需要哪些参数?
16. 在当前光标位置显示字符并设置属性时需要哪些参数?
17. INT 10h 的哪个功能切换闪烁/亮度模式?
18. 使用 INT 10h 功能 6 清除屏幕时 AH 和 AL 中应放入什么值?
19. 挑战: 如果你有一条狗, 你觉得它为什么会对你接连数小时盯着灰色的计算机屏幕而感到吃惊?

15.4 使用 INT 10h 绘图

INT 10h 的功能 0Ch 在图形模式下绘制一个像素点。可以使用该功能绘制复杂的图形和线条，不过速度慢得难以忍受。我们以该功能为起点学习绘图的基础知识，后面再演示如何通过直接写视频 RAM 绘制图形。

在显卡处于图形模式时可使用 INT 10h 的功能 9h 在屏幕上绘制文本。

在绘制像素点之前，必须把视频适配卡设置为表 15.7 中的某种标准图形模式，使用 INT 10h 功能 0（设置视频模式）可以做到这一点。

表 15.7 INT 10h 可以识别的视频图形模式

模 式	分辨率（列×行，单位是像素）	颜 色 数
6	640×200	2
0Dh	320×200	16
0Eh	640×350	16
0Fh	640×350	2
10h	640×200	16
11h	640×480	2
12h	640×480	16
13h	320×200	256
6Ah	800×600	16

坐标：每种视频模式的分辨率都是用 $XMax \times YMax$ 的形式表示的， $XMax$ 代表水平行最多可显示的像素数， $YMax$ 代表垂直行最多可显示的像素数。屏幕坐标范围从左上角的 $x = 0, y = 0$ 到右下角的 $x = Xmax - 1, y = Ymax - 1$ 。

15.4.1 和像素相关的 INT 10h 功能调用

写像素 (0Ch)

如下表所示，在视频控制器处于图形模式时，INT 10h 的功能 0Ch 在屏幕上绘制一个像素点。功能 0Ch 执行得相当慢，这在绘制大量像素时更加明显。大多数图形应用程序在计算完每个像素的颜色数、水平分辨率等数据后，直接写显存。

INT 10h 功能 0Ch

描述	写像素
接收参数	AH = 0Ch AL = 像素值 BH = 视频页 CX = x 坐标 DX = y 坐标
返回值	无
调用示例	<pre> mov ah, 0Ch mov al, pixelValue mov bh, videoPage mov cx, x_coord mov dx, y_coord int 10h </pre>
注意	视频显示必须处于图形模式下。像素值的范围和坐标范围与当前的图形模式有关。如果 AL 的位 7 置位，新的像素同当前像素的内容进行异或运算（允许像素被删除）

读像素 (0Dh)

如下表所示, 功能 0Dh 读取屏幕上指定坐标位置处的像素值并在 AL 中返回像素值。

INT 10h 功能 0Dh	
描述	读像素
接收参数	AH = 0Dh BH = 视频页 CX = x 坐标 DX = y 坐标
返回值	AL = 像素值
调用示例	<pre> mov ah, 0Dh mov bh, 0 ; 视频页 0 mov cx, x_coord mov dx, y_coord int 10h mov pixelValue, al </pre>
注意	视频显示必须处于图形模式下。像素值的范围和坐标范围与当前的图形模式有关

15.4.2 画线程序

画线 (DrawLine) 程序使用 INT 10h 切换到图形模式下并绘制一条水平直线。如果在 MS-Windows 下运行该程序, 应该通过 Alt-Enter 按键切换到全屏模式^①。下面是完整的程序清单:

```

TITLE DrawLine Program                      (DrawLine.asm)
; This program draws text and a straight line in graphics mode.
INCLUDE Irvine16.inc
;----- 视频模式常量 -----
Mode_06 = 6; 640 X 200,  2 colors
Mode_0D = 0Dh; 320 X 200, 16 colors
Mode_0E = 0Eh; 640 X 200, 16 colors
Mode_0F = 0Fh; 640 X 350,  2 colors
Mode_10 = 10h; 640 X 350, 16 colors
Mode_11 = 11h; 640 X 480,  2 colors
Mode_12 = 12h; 640 X 480, 16 colors
Mode_13 = 13h; 320 X 200, 256 colors
Mode_6A = 6Ah; 800 X 600, 16 colors

.data
saveMode  BYTE  ?           ; 保存当前视频模式
currentX  WORD  100         ; 列号 (X 坐标)
currentY  WORD  100         ; 行号 (Y 坐标)
COLOR     = 1001b          ; 行的颜色 (青)

progTitle BYTE "DrawLine.asm"
TITLE_ROW = 5
TITLE_COLUMN = 14

; 当使用双色模式时, 将 COLOR 设为 1 (白色)
.code

```

① 在显存较少的计算机上, 在 MS-Windows 下运行 Pixel1.asm 和 Pixel2.asm 或许会有问题。如果是这样, 可以切换到另外一种模式 (如模式 11h) 或重新引导到纯 MS-DOS 下再运行程序。

```

main PROC
    mov     ax,@data
    mov     ds,ax

; 保存当前的视频模式
    mov     ah,0Fh
    int     10h
    mov     saveMode,al

; 改成图形模式
    mov     ah,0                ; 设置视频模式
    mov     al,Mode_6A
    int     10h

; 以文本方式写程序名
    mov     ax,SEG progTitle    ; 获取 progTitle 段
    mov     es,ax              ; 保存在 ES 中
    mov     bp,OFFSET progTitle
    mov     ah,13h             ; 功能: 写字符串
    mov     al,0               ; 模式: 只有字符代码
    mov     bh,0               ; 视频页 0
    mov     bl,7               ; 属性 = normal
    mov     cx,SIZEOF progTitle ; 字符串长度
    mov     dh,TITLE_ROW       ; 行 (字符数)
    mov     dl,TITLE_COLUMN     ; 列 (字符数)
    int     10h

; 画一条直线
    LineLength = 100

    mov     dx,currentY
    mov     cx,LineLength      ; 循环计数器

L1:
    push    cx
    mov     ah,0Ch             ; 写像素
    mov     al,COLOR           ; 像素颜色
    mov     bh,0               ; 视频页 0
    mov     cx,currentX
    int     10h
    inc     currentX
    ; inc    color              ; 启用查看多颜色行的功能
    pop     cx
    Loop    L1

; 等待键击
    mov     ah,0
    int     16h

; 恢复开始的视频模式
    mov     ah,0                ; 设置视频模式
    mov     al,saveMode         ; 保存的视频模式
    int     10h
    exit
main ENDP
END main

```

改变视频模式: 读者可以尝试不同的视频模式, 只需修改当前选择模式 6Ah 的那条语句即可:

```

mov     ah,0                ; 设置视频模式
mov     al,Mode_6A          ; 修改为不同的模式
int     10h                 ; 调用 BIOS 例程

```

15.4.3 笛卡儿坐标程序

笛卡儿坐标 (Cartesian Coordinate) 程序绘制笛卡儿坐标系的 X 轴和 Y 轴, 其中原点位于 X =


```

main endp

;-----
DrawHorizLine PROC
;
; Draws a horizontal line starting at position X,Y with
; a given length and color.
; Receives: CX = X-coordinate, DX = Y-coordinate,
;           AX = length, and BL = color
; Returns: nothing
;-----
.data
currX WORD ?

.code
    pusha
    mov  currX,cx          ; 保存 X 坐标
    mov  cx,ax             ; 循环计数器

DHL1:
    push cx                ; 保存循环计数器
    mov  al,bl             ; 颜色
    mov  ah,0Ch            ; 绘制像素
    mov  bh,0              ; 视频页
    mov  cx,currX          ; 取得 X 坐标
    int  10h
    inc  currX              ; 向右移动一个像素
    pop  cx                ; 恢复循环计数器
    Loop DHL1

    popa
    ret
DrawHorizLine ENDP

;-----
DrawVerticalLine PROC
;
; Draws a vertical line starting at position X,Y with
; a given length and color.
; Receives: CX = X-coordinate, DX = Y-coordinate,
;           AX = length, BL = color
; Returns: nothing
;-----
.data
currY WORD ?

.code
    pusha
    mov  currY,dx          ; 保存 Y 坐标
    mov  currX,cx          ; 保存 X 坐标
    mov  cx,ax             ; 循环计数器

DVL1:
    push cx                ; 保存循环计数器
    mov  al,bl             ; 颜色
    mov  ah,0Ch            ; 功能: 绘制像素
    mov  bh,0              ; 设置视频页
    mov  cx,currX          ; 设置 X 坐标

```

```

mov dx,currY          ; 设置 Y 坐标
int 10h               ; 绘制像素
inc currY             ; 向下移动一个像素
pop cx                ; 恢复循环计数器
Loop DVL1

popa
ret
DrawVerticalLine ENDP
END main

```

15.4.4 把笛卡儿坐标转换为屏幕坐标

笛卡儿坐标系中的点并不和 BIOS 图形系统使用的绝对坐标相对应。在前面的两个例子中,很明显屏幕的坐标从左上角的 $sx=0, sy=0$ 开始。 sx 的值向右边递增, sy 的值向屏幕底端递增。可以使用下面的公式把笛卡儿坐标 (X, Y) 转换为屏幕坐标 (sx, sy) :

$$sx = (sOrigX + X) \quad sy = (sOrigY - Y)$$

其中 $sOrigX$ 和 $sOrigY$ 是笛卡儿坐标系原点的屏幕坐标。在笛卡儿坐标系程序中 (见 15.4.3 节), 我们使用 $sOrigX=400$ 和 $sOrigY=300$ 把原点放在了屏幕的中央。图 15.3 中使用 4 个点测试上面的公式, 表 15.8 给出了计算的结果。

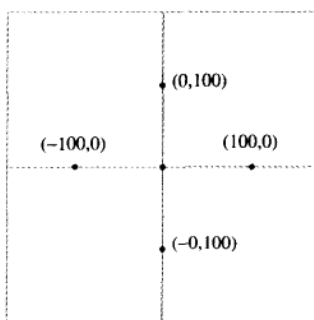


图 15.3 测试转换公式的坐标

表 15.8 测试转换公式

笛卡儿坐标 (X, Y)	$(400 + X, 300 - Y)$	屏幕坐标 (sx, sy)
(0,100)	$(400+0, 300-100)$	(400,200)
(100,0)	$(400+100, 300-0)$	(500,300)
(0,-100)	$(400+0, 300-(-100))$	(400,400)
(-100,0)	$(400+(-100), 300-0)$	(300,300)

15.4.5 本节习题

1. INT 10h 的哪个功能调用在视频显示上绘制一个像素点?
2. 在使用 INT 10h 绘制像素点时, 在 AL, BH, CX 和 DX 寄存器中需要放入什么值?
3. 使用 INT 10h 绘制像素点的主要缺点是什么?
4. 写汇编语句, 把视频显示卡设置为模式 11h。
5. 哪种视频模式的分辨率是 800×600 像素, 16 色?
6. 把笛卡儿坐标系中的 X 坐标转换成屏幕像素的 X 坐标的公式是什么 (使用 sx 代表屏幕

列坐标, 使用 sOrigX 代表笛卡儿坐标系原点的 X 坐标) ?

7. 如果笛卡儿坐标系的原点位于屏幕坐标位置 $sy = 250$, $sx = 350$ 处, 把下面的 (X, Y) 形式的笛卡儿坐标系中的点转换成屏幕坐标 (sx, sy) :
 - a. $(0, 100)$
 - b. $(25, 25)$
 - c. $(-200, -150)$

15.5 内存映射图形

除了一些非常简单的图形输出任务之外, 使用 INT 10h 绘制像素的方法慢得令人难以忍受, BIOS 每次绘制一个像素时都要执行大量的代码。本节讲述如何通过输入输出端口把图形数据直接写入显存, 这种方法更加高效, 正如一些专业软件所做的那样。

15.5.1 模式 13h: 320×200 , 256 色

对于内存映射图形来说, 视频模式 13h 最容易使用。这时屏幕像素映射为一个二维字节数组, 其中每个像素单独占用一个字节。数组从屏幕左上角的像素开始, 屏幕顶端的一行占用 320 字节, 在偏移 320 处的字节映射为第二行的第一个像素, 其余行以类似的方式映射。数组中的最后一个字节映射为屏幕右下角的像素。为什么每个像素要占用一整个字节呢? 这是因为总共有 256 种颜色, 每个字节都要指示一种不同的颜色。

OUT 指令: 像素和颜色值使用 OUT 指令 (输出到端口) 送到视频适配器, 其中 DX 中存放 16 位的端口值, 要发送的值存放在 AL, AX 或 EAX 中。例如, 视频色彩调色板位于端口地址 3C8h 处, 下面的指令把值 20h 送至该端口:

```
mov dx, 3C8h          ; 端口地址
mov al, 20h           ; 要输出的值
out dx, al            ; 发送值至端口
```

色彩索引: 在模式 13h 中最有趣的事情是每个整数色彩值并不直接表示一种颜色, 相反, 该整数值表示一个称为调色板 (见图 15.4) 的色彩表的索引。调色板中的每个项都由三个独立的整数 (0~63) 构成, 称为 RGB 值 (红、绿、蓝)。调色板的第 0 项控制着屏幕的背景颜色。

使用这种方案可以创建 $262\,144$ (64^3) 种不同的色彩, 但在某一时刻只能显示 256 种不同的色彩, 程序在运行的时候可非常容易地修改调色板以改变色彩。现代操作系统如 Windows 和 Linux 都提供了 (至少) 24 位的色彩, 每个 RGB 值的范围都是 0~255, 这种方案提供了 256^3 (1670 万) 种不同的颜色。

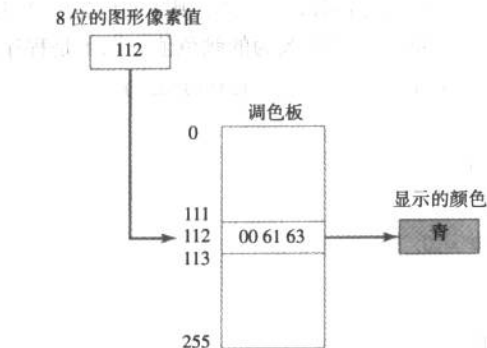


图 15.4 把像素色彩索引转换为要显示的颜色

RGB 色彩

RGB 色彩依据的是光的叠加混合（相加色），与混合液体颜料使用的减色正好相反。例如，使用叠加混合法把所有色彩的亮度设为 0 就可以创建黑色。白色则正好相反，它是把所有的色彩亮度级都设为 63（最大值）形成的。事实上，如下表所示，当三种亮度相等的时候，就得到了各种灰度不同的颜色。

红	绿	蓝	颜色
0	0	0	黑
20	20	20	深灰
35	35	35	灰
50	50	50	浅灰
63	63	63	白

把其他任何两种色彩亮度都设为 0 即可创建纯色。要获得亮色，可以等值增加另外两种颜色。

红	绿	蓝	颜色
63	0	0	亮红
10	0	0	深红
30	0	0	红
63	40	40	粉

亮蓝、深蓝、浅蓝和亮绿、深绿、浅绿等颜色可以用类似的方法创建。当然，还可以混合两种不同的色彩来创建如洋红和淡紫之类的颜色。举例如下。

红	绿	蓝	颜色
0	30	30	青
30	30	0	黄
30	0	30	洋红
40	0	63	淡紫

15.5.2 内存映射图形程序

下面给出的内存映射图形程序在模式 13h 下使用直接内存映射的方法在屏幕上绘制包含 10 个像素的行。主程序调用子过程把视频模式设置为 13h，设置屏幕的背景色，绘制一些彩色像素，然后把显卡恢复到初始的视频模式。有两个输出端口用于控制视频调色板：送往端口 3C8h 的值表示要修改的调色板表项，送往端口 3C9h 的是要修改为的颜色值。以下是程序清单：

```
; Memory Mapped Graphics, Mode 13      (Mode13.asm)
INCLUDE Irvine16.inc
VIDEO_PALLETE_PORT = 3C8h
COLOR_SELECTION_PORT = 3C9h
COLOR_INDEX = 1
PALLETE_INDEX_BACKGROUND = 0
SET_VIDEO_MODE = 0
GET_VIDEO_MODE = 0Fh
VIDEO_SEGMENT = 0A000h
WAIT_FOR_KEYSTROKE = 10h
MODE_13 = 13h
```

```

.data
saveMode BYTE ?           ; 保存的视频模式
xVal WORD ?               ; X 坐标
yVal WORD ?               ; Y 坐标
msg BYTE "Welcome to Mode 13!",0

.code
main PROC
    mov     ax,@data
    mov     ds,ax

    call    SetVideoMode
    call    SetScreenBackground

; 显示一条欢迎信息
    mov     edx,OFFSET msg
    call    WriteString

    call    Draw_Some_Pixels
    call    RestoreVideoMode
    exit
main ENDP

;-----
SetScreenBackground PROC
;
; Sets the screen's background color. Video
; palette index 0 is the background color.
;-----
    mov     dx,VIDEO_PALLETE_PORT
    mov     al,PALLETE_INDEX_BACKGROUND
    out     dx,al

; 把屏幕的背景设为深蓝
    mov     dx,COLOR_SELECTION_PORT
    mov     al,0           ; 红
    out     dx,al
    mov     al,0           ; 绿
    out     dx,al
    mov     al,35          ; 蓝 (亮度 35/63)
    out     dx,al

    ret
SetScreenBackground endp

;-----
SetVideoMode PROC
;
; Saves the current video mode, switches to a
; new mode, and points ES to the video segment.
;-----
    mov     ah,GET_VIDEO_MODE
    int     10h
    mov     saveMode,al     ; 保存之

    mov     ah,SET_VIDEO_MODE
    mov     al,MODE_13      ; 至模式 13h
    int     10h

    push    VIDEO_SEGMENT   ; 视频段地址
    pop     es              ; ES 指向视频段

    ret
SetVideoMode ENDP
;-----

```



```

RestoreVideoMode PROC
;
; Waits for a key to be pressed and restores
; the video mode to its original value.
;-----
    mov     ah, WAIT_FOR_KEYSTROKE
    int     16h
    mov     ah, SET_VIDEO_MODE      ; 重设视频模式
    mov     al, saveMode            ; 至已保存的模式
    int     10h
    ret
RestoreVideoMode ENDP

;-----
Draw_Some_Pixels PROC
;
; Sets individual palette colors and draws
; several pixels.
;-----
; 把索引 1 处的颜色改为白色 (63, 63, 63)
    mov     dx, VIDEO_PALLETE_PORT
    mov     al, 1                   ; 设置调色板索引 1
    out     dx, al

    mov     dx, COLOR_SELECTION_PORT
    mov     al, 63                  ; 红
    out     dx, al
    mov     al, 63                  ; 绿
    out     dx, al
    mov     al, 63                  ; 蓝
    out     dx, al

; 计算第一个像素的视频缓冲区偏移地址
; 方法是与模式 13h 特别相关的, 分辨率是 320*200
    mov     xVal, 160               ; 屏幕中心
    mov     yVal, 100
    mov     ax, 320                 ; 320 用于视频模式 13h
    mul     yVal                   ; Y 坐标
    add     ax, xVal                ; x 坐标

; 把色彩索引值放到视频缓冲区中
    mov     cx, 10                  ; 绘制 10 个像素
    mov     di, ax                  ; AX 包含缓冲区的偏移地址
; Draw the pixels now. By default, the assembler assumes
; DI is an offset from the segment address in DS. The
; segment override ES:[DI] tells the CPU to use the segment
; address in ES instead. ES currently points to VRAM.
DP1:
    mov     BYTE PTR es:[di], COLOR_INDEX
    add     di, 5                   ; 向右移动 5 个像素
    loop    DP1

    ret
Draw_Some_Pixels ENDP
END main

```

由于所有的像素恰好在同一行上, 因此程序相当容易实现。若要绘制垂直直线, 可向 DI 加 320 以移动到下一行的像素点上。或者, 绘制倾斜度为一个像素的斜线可给 DI 加 361。绘制任意两点之间的直线最好使用 Bresenham 算法, 这在很多网站上都有详细的介绍。

15.5.3 本节习题

1. (真/假): 视频模式 13h 把屏幕像素映射为二维字节数组, 每字节对应两个像素。
2. (真/假): 在视频模式 13h 下, 每个屏幕行使用 320 字节的存储空间。
3. 用一句话说明视频 13h 模式是如何设置像素的色彩的。
4. 在视频模式 13h 下色彩索引是如何使用的?
5. 在视频模式 13h 下, 调色板的每个项包含的内容是什么?
6. 暗灰色包含的三个 RGB 值分别是什么?
7. 白色包含的三个 RGB 值分别是什么?
8. 亮红色包含的三个 RGB 值分别是什么?
9. 挑战: 如何在视频模式 13h 下把屏幕背景色设置为绿色。
10. 挑战: 如何在视频模式 13h 下把屏幕背景色设置为白色。

15.6 鼠标程序设计

鼠标通常通过 RS-232 串口、PS/2 鼠标口、USB 端口或无线连接同计算机的主板相连。MS-DOS 要求在内存中安装一个驱动程序才能探测鼠标, MS-Windows 有自己内建的鼠标驱动, 但现在我们着重讲述 MS-DOS 提供的功能。

鼠标的移动是以称为 mickey (猜猜这个名字是如何得来的?) 的单位跟踪计量的, 一个 mickey 代表鼠标实际物理移动大约 1/200 英寸 (1 英寸 = 2.54 cm)。mickey 到像素的转换比率可以进行设置, 默认情况下是 8 个水平像素对应 8 个 mickey, 8 个垂直像素对应 16 个 mickey^①。鼠标的移动有一个双速限制, 默认是 64 mickey/s。

15.6.1 鼠标 INT 33h 功能调用

INT 33h 提供了关于鼠标的信息, 包括鼠标的当前位置、最后按下的按键和移动速度等。INT 33h 还可以用来隐藏和显示鼠标光标。本章讲述一些最基本的鼠标功能。INT 33h 在 AX 中而不是 AH 中接收功能号 (AH 在 BIOS 功能中常用)。

复位和获取鼠标状态

INT 33h 功能 0 把鼠标复位并确认鼠标可用。如果发现了鼠标, 则把鼠标定位在屏幕的中央, 其显示页被设置为视频页 0, 鼠标指针被隐藏, mickey 到像素的转换比率和鼠标移动速度被设置为默认值, 鼠标的移动范围被设置为整个屏幕。细节如下表所示。

INT 33h 功能 0	
描述	复位鼠标并获取状态
接收参数	AX = 0
返回值	如果鼠标可用, 则 AX = FFFFh, BX = 鼠标键数目; 否则 AX = 0
调用示例	<pre> mov ax, 0 int 33h cmp ax, 0 je MouseNotAvailable mov numberOfButtons, bx </pre>
注意	如果在调用该功能之前鼠标可见, 该功能调用将隐藏鼠标

① 来自 Ray Duncan 的 *Advanced MS-DOS Programming*, 第二版, 1998, Microsoft 出版社, 第 601 页

显示和隐藏鼠标指针

下面两个表所示的 INT 33h 功能 1 和功能 2 专门用于显示和隐藏鼠标指针。鼠标驱动内部保留着一个计数值，调用功能 1 时该计数值增 1（在非 0 的情况下），调用功能 2 时该计数值减 1。当计数值为非负值时鼠标指针显示，功能 0（复位鼠标）把该计数值设置为-1。

INT 33h 功能 1	
描述	显示鼠标指针
接收参数	AX = 1
返回值	无
调用示例	<pre>mov ax, 1 int 33h</pre>
注意	鼠标驱动保留着一个该功能调用次数的计数值，该功能调用使得该计数值增 1

INT 33h 功能 2	
描述	隐藏鼠标指针
接收参数	AX = 2
返回值	无
调用示例	<pre>mov ax, 2 int 33h</pre>
注意	隐藏之后鼠标驱动会继续跟踪鼠标的位置，该功能调用使得驱动内部的计数值减 1

获取鼠标位置和状态

INT 33h 功能 3 获取鼠标的位置和状态，具体如下表所示。

INT 33h 功能 3	
描述	获取鼠标位置和状态
接收参数	AX = 3
返回值	BX = 鼠标键状态 CX = X 坐标（以像素为单位） DX = Y 坐标（以像素为单位）
调用示例	<pre>mov ax, 3 int 33h test bx, 1 jne Left_Button_Down test bx, 2 jne Right_Button_Down test bx, 4 jne Center_Button_Down mov Xcoord, cx mov YCoord, dx</pre>
注意	BX 中返回的鼠标键状态的含义如下：位 0 置位表示左键按下；位 1 置位表示右键按下；位 2 置位表示中键按下

把像素坐标转换为字符坐标：MS-DOS 中的标准文本字体是 8 像素宽、16 像素高。因此像素坐标除以字符的大小就可以转换成字符坐标。假设像素坐标和字符坐标都是从 0 开始计算的，下面的公式把像素坐标 P 转换成字符坐标 C ，字符的大小用 D 表示：

$$C = \text{int}(P/D)$$

例如, 假设字符是 8 像素宽的, 如果 INT 33h 功能 3 返回的 X 坐标是 100 (像素), 那么转换为字符坐标位置就是 12, 因为 $C = \text{int}(100 / 8)$ 。

设置鼠标位置

INT 33h 功能 4 把鼠标定位到指定的 X 和 Y 像素坐标处, 如下表所示。

INT 33h 功能 4	
描述	设置鼠标位置
接收参数	AX = 4 CX = X 坐标 (以像素为单位) DX = Y 坐标 (以像素为单位)
返回值	无
调用示例	<pre> mov ax, 4 mov cx, 200 ; X 坐标 mov dx, 100 ; Y 坐标 int 33h </pre>
注意	如果该位置位于排除显示区内, 则鼠标不显示

把字符坐标转换为像素坐标: 可使用下面的公式把字符坐标转换为像素坐标, 其中 C 为字符坐标, P 为像素坐标, D 为字符大小:

$$P = C \times D$$

在水平方向上, P 是字符单元的左边所在的像素位置, 在垂直方向上, P 是字符单元的顶端所在的像素位置。如果字符宽度是 8 个像素并且想把鼠标定位在第 12 个字符单元的左边缘处, 那么字符单元最左边的像素的 X 坐标是 96。

获取按钮的按下和释放

功能 5 返回鼠标按键的状态以及鼠标键最后按下的位置。在一个事件驱动的编程环境中, 拖动事件总是以鼠标键的按下开始的。一旦对某个鼠标键调用了该功能, 那么该按键的状态就被重置了, 第二次调用该功能时什么也不会返回。

INT 33h 功能 5	
描述	获取按钮的按下信息
接收参数	AX = 5 BX = 按键 ID (0 = 左, 1 = 右, 2 = 中)
返回值	AX = 键的状态 BX = 鼠标按键下压计数 CX = 最后按下键的 X 坐标 DX = 最后按下键的 Y 坐标
调用示例	<pre> mov ax, 5 mov bx, 0 ; 鼠标按键 ID int 33h test ax, 1 ; 左键按下? jz skip ; 否: 跳过 mov X_coord, cx ; 是: 保存坐标 mov Y_coord, dx </pre>
注意	AX 中返回的鼠标按键状态含义如下: 位 0 置位表示左键按下; 位 1 置位表示右键按下; 位 2 置位表示中键按下

如下表所示, 功能 6 获取鼠标释放的信息。在事件驱动的程序设计中, 鼠标单击事件只在鼠标释放时发生, 拖动事件在鼠标释放时结束。

INT 33h 功能 6	
描述	获取鼠标释放的信息
接收参数	AX = 6 BX = 按键 ID (0 = 左, 1 = 右, 2 = 中)
返回值	AX = 键的状态 BX = 鼠标按键释放计数 CX = 最后释放键的 X 坐标 DX = 最后释放键的 Y 坐标
调用示例	<pre> mov ax, 6 mov bx, 0 ; 鼠标按键 ID int 33h test ax, 1 ; 左键释放? jz skip ; 否: 跳过 mov X_coord, cx ; 是: 保存坐标 mov Y_coord, dx </pre>
注意	AX 中返回的鼠标按键状态含义如下: 位 0 置位表示左键被释放; 位 1 置位表示右键被释放; 位 2 置位表示中键被释放

设置垂直和水平限制

如下面两个表所示, INT 33h 功能 7 和功能 8 允许设置鼠标在屏幕上移动的范围限制, 读者可以设置鼠标光标的最大和最小坐标。如果需要的话, 要相应移动鼠标指针使之位于新的范围之内。

INT 33h 功能 7	
描述	设置水平限制
接收参数	AX = 7 CX = 最小的 X 坐标 (以像素为单位) DX = 最大的 X 坐标 (以像素为单位)
返回值	无
调用示例	<pre> mov ax, 7 mov cx, 100 ; 设置 x 的范围至 mov dx, 700 ; (100, 700) int 33h </pre>

INT 33h 功能 8	
描述	设置垂直限制
接收参数	AX = 8 CX = 最小的 Y 坐标 (以像素为单位) DX = 最大的 Y 坐标 (以像素为单位)
返回值	无
调用示例	<pre> mov ax, 8 int 33h mov cx, 100 ; 设置 y 的范围至 mov dx, 500 ; (100, 500) int 33h </pre>

其他鼠标功能

很多 INT 33h 的功能在配置和控制鼠标的行为时非常有用，它们列在表 15.9 中，但本书中没有足够的篇幅来详细讲述这些功能。

表 15.9 其他鼠标功能

功 能	描 述	输入/输出参数
AX = 0Fh	设置 mickey 到像素的转换比率 (每 8 个像素)	接收: CX = 水平 mickey 数, DX = 垂直 mickey 数。默认值是 CX = 8, DX = 16
AX = 10h	设置鼠标排除区(阻止鼠标进入一个矩形区域)	接收: CX, DX = 左上角的 X, Y 坐标, SI, DI = 右下角的 X, Y 坐标
AX = 13h	设置双速限制	接收: DX = 限制速度, 单位是 mickey/s (默认值是 64)
AX = 1Ah	设置鼠标的敏感度和限制	接收: BX = 水平速度 (mickey/s) CX = 垂直速度 (mickey/s) DX = 以 mickey/s 为单位的双速限制
AX = 1Bh	获取鼠标的敏感度和限制	返回: BX = 水平速度 CX = 垂直速度 DX = 双速限制
AX = 1Fh	禁止鼠标驱动	返回: 如果成功, AX = FFFFh
AX = 20h	允许鼠标驱动	无
AX = 24h	获取鼠标信息	错误返回 FFFFh; 否则返回: BH = 主版本号, BL = 次版本号, CH = 鼠标类型 (1 = 总线, 2 = 串口, 3 = 输入口, 4 = PS/2, 5 = HP); CL = IRQ 号 (PS/2 鼠标是 0)

15.6.2 鼠标跟踪程序

我们编写了一个简单的鼠标跟踪程序跟踪文本鼠标光标的移动。鼠标的 X 和 Y 坐标随着鼠标的移动在屏幕右下角不断更新，用户按下左键时，鼠标的位置显示在屏幕的左下角。下面是程序清单：

```

TITLE Tracking the Mouse                                (mouse.asm)

; Demonstrates basic mouse functions available via INT 33h.
; In Standard DOS mode, each character position in the DOS
; window equals 8 mouse units.

INCLUDE Irvine16.inc

GET_MOUSE_STATUS = 0
SHOW_MOUSE_POINTER = 1
HIDE_MOUSE_POINTER = 2
GET_CURSOR_SIZE = 3
GET_BUTTON_PRESS_INFO = 5
GET_MOUSE_POSITION_AND_STATUS = 3
ESCkey = 1Bh

.data
greeting    BYTE "[Mouse.exe] Press Esc to quit",0
statusLine  BYTE "Left button: "
            BYTE "Mouse position: ",0
blanks      BYTE " ",0
xCoord      WORD 0 ; current X-coordinate
yCoord      WORD 0 ; current Y-coordinate

```

```

xPress WORD 0                ; X-coord of last button press
yPress WORD 0                ; Y-coord of last button press

; Display coordinates.
statusRow    BYTE ?
statusCol    BYTE 15
buttonPressCol BYTE 20
statusCol2   BYTE 60
coordCol     BYTE 65

.code
main PROC
    mov     ax,@data
    mov     ds,ax
    call    clrscr

; Get the screen X/Y coordinates.
    call    GetMaxXY          ; DH = rows, DL = columns
    dec     dh                ; calculate status row value
    mov     statusRow,dh

; Hide the text cursor and display the mouse.
    call    HideCursor
    mov     dx,OFFSET greeting
    call    WriteString
    call    ShowMousePointer

; Display status information on the bottom screen line.
    mov     dh,statusRow
    mov     dl,0
    call    Gotoxy
    mov     dx,OFFSET statusLine
    call    WriteString

; Loop: show mouse coordinates, check for left mouse
; button press or keypress (Esc key).
L1:  call    ShowMousePosition
    call    LeftButtonPress    ; check for button press
    mov     ah,11h             ; key pressed already?
    int     16h
    jz      L2                 ; no, continue the loop
    mov     ah,10h             ; remove key from buffer
    int     16h
    cmp     al,ESCKey          ; yes. Is it the ESC key?
    je      quit               ; yes, quit the program
L2:  jmp     L1                 ; no, continue the loop

; Hide the mouse, restore the text cursor, clear
; the screen, and wait for a key press.
quit:
    call    HideMousePointer
    call    ShowCursor
    call    clrscr
    call    WaitMsg
    exit
main ENDP

;-----
GetMousePosition PROC USES ax
;
; Gets the current mouse position and button status.
; Receives: nothing
; Returns:  BX = button status (0 = left button down,
;           (1 = right button down, 2 = center button down)
;           CX = X-coordinate
;

```

```

;          DX = Y-coordinate
;-----
    mov     ax,GET_MOUSE_POSITION_AND_STATUS
    int     33h
    ret
GetMousePosition ENDP

;-----
HideCursor PROC USES ax cx
;
; Hide the text cursor by setting its top line
; value to an illegal value.
; Receives: nothing. Returns: nothing
;-----
    mov     ah,GET_CURSOR_SIZE
    int     10h
    or      ch,30h                ; set upper row to illegal value
    mov     ah,1                  ; set cursor size
    int     10h
    ret
HideCursor ENDP

;-----
ShowCursor PROC USES ax cx
;
; Show the text cursor by setting size to default.
; Receives: nothing. Returns: nothing
;-----
    mov     ah,GET_CURSOR_SIZE
    int     10h
    mov     ah,1                  ; set cursor size
    mov     cx,0607h              ; default size
    int     10h
    ret
ShowCursor ENDP

;-----
HideMousePointer PROC USES ax
;
; Hides the mouse pointer.
; Receives: nothing. Returns: nothing
;-----
    mov     ax,HIDE_MOUSE_POINTER
    int     33h
    ret
HideMousePointer ENDP

;-----
ShowMousePointer PROC USES ax
;
; Makes the mouse pointer visible.
; Receives: nothing. Returns: nothing
;-----
    mov     ax,SHOW_MOUSE_POINTER ; make mouse cursor visible
    int     33h
    ret
HideMousePointer ENDP

;-----
ShowMousePointer PROC USES ax
;
; Makes the mouse pointer visible.
; Receives: nothing. Returns: nothing
;-----

```



```

        mov     ax,SHOW_MOUSE_POINTER ; make mouse cursor visible
        int     33h
        ret
ShowMousePointer ENDP

;-----
LeftButtonPress PROC
;
; Checks for the most recent left mouse button press
; and displays the mouse location.
; Receives: nothing. Returns: nothing
;-----
        pusha
        mov     ax,GET_BUTTON_PRESS_INFO
        mov     bx,0                ; specify the left button
        int     33h

; Exit proc if the coordinates have not changed.
        cmp     cx,xPress           ; same X coordinate?
        jne     L1                  ; no: continue
        cmp     dx,yPress           ; same Y coordinate?
        je      L2                  ; yes: exit

; Coordinates have changed, so save them.
L1:  mov     xPress,cx
     mov     yPress,dx

; Position the cursor, clear the old numbers.
     mov     dh,statusRow           ; screen row
     mov     dl,statusCol           ; screen column
     call    Gotoxy
     push    dx
     mov     dx,OFFSET blanks
     call    WriteString
     pop     dx

; Show coordinates where mouse button was pressed.
     call    Gotoxy
     mov     ax,xCoord
     call    WriteDec
     mov     dl,buttonPressCol
     call    Gotoxy
     mov     ax,yCoord
     call    WriteDec

L2:  popa
     ret
LeftButtonPress ENDP

;-----
SetMousePosition PROC
;
; Set the mouse's position on the screen.
; Receives: CX = X-coordinate
;           DX = Y-coordinate
; Returns: nothing
;-----
        mov     ax,4
        int     33h
        ret
SetMousePosition ENDP

;-----
ShowMousePosition PROC
;
; Get and show the mouse coordinates at the

```

```

; bottom of the screen.
; Receives: nothing
; Returns: nothing
;-----
        pusha
        call  GetMousePosition

; Exit proc if the coordinates have not changed.
        cmp  cx,xCoord          ; same X coordinate?
        jne  L1                 ; no: continue
        cmp  dx,yCoord          ; same Y coordinate?
        je   L2                 ; yes: exit

; Save the new X and Y coordinates.
L1:  mov  xCoord,cx
     mov  yCoord,dx

; Position the cursor, clear the old numbers.
     mov  dh,statusRow          ; screen row
     mov  dl,statusCol2         ; screen column
     call  Gotoxy
     push  dx
     mov  dx,OFFSET blanks
     call  WriteString
     pop  dx

; Show the mouse coordinates.
     call  Gotoxy
     mov  ax,xCoord
     call  WriteDec
     mov  dl,coordCol           ; screen column
     call  Gotoxy
     mov  ax,yCoord
     call  WriteDec

L2:  popa
     ret
ShowMousePosition ENDP
END main

```

不同的行为：由于下面两个因素的不同，程序的行为可能也会有所不同：（1）运行的 Windows 的版本；（2）在控制台窗口下运行还是在全屏幕模式下运行。例如，在 Windows XP 下，控制台窗口默认是 50 行文本。在全屏幕模式下运行时，鼠标光标是一个实心方块，其坐标每次改变一个像素，鼠标光标只有在水平移动了 8 个像素或垂直移动了 16 个像素后才从一个字符跳到下一个字符处。在控制台窗口模式下，鼠标光标是一个指针，其坐标每次水平改变 8 个像素、垂直改变 16 个像素。

15.6.3 本节习题

1. INT 33h 的哪个功能复位鼠标并且获取鼠标的状态？
2. 写汇编语句，复位鼠标并且获取鼠标的状态。
3. INT 33h 的哪些功能显示和隐藏鼠标指针？
4. 写汇编语句，显示和隐藏鼠标指针。
5. INT 33h 的哪个功能获取鼠标的位置和状态？
6. 写汇编语句，获取鼠标的位置并存储在变量 mouseX 和 mouseY 中。
7. INT 33h 的哪个功能设置鼠标的位置？
8. 写汇编语句把鼠标指针的位置设置为 $X = 100$ ， $Y = 400$ 。

9. INT 33h 的哪个功能获取鼠标按钮按下的信息？
10. 写汇编语句，当鼠标左键按下时跳转到标号 Button1 处。
11. INT 33h 的哪个功能获取鼠标按钮释放的信息？
12. 写汇编语句，在鼠标右键释放时获取鼠标的位置，并把位置存储在变量 mouseX 和 mouseY 中。
13. 写汇编语句，把鼠标的垂直限制设置为 200 和 400。
14. 写汇编语句，把鼠标的水平限制设置为 300 到 600。
15. 挑战：假设想要鼠标指针指向文本模式下行 10、列 20 处字符单元的左上角，给 INT 33h 功能 4 传递的 X 和 Y 的值应该是什么？假设每个字符水平 8 个像素宽，垂直 16 个像素高。
16. 挑战：假设想要鼠标指针指向文本模式下行 15、列 21 处字符单元的中央，给 INT 33h 功能 4 传递的 X 和 Y 的值应该是什么？假设每个字符水平 8 个像素宽，垂直 16 个像素高。
17. 挑战：是谁发明了鼠标？在什么时间（哪一年）？什么地点？

15.7 本章小结

与工作 MS-DOS 级相比，工作在 BIOS 级可以对计算机的输入输出设备拥有更多的控制。本章展示了如何使用 INT 16h 对键盘编程、使用 INT 10h 对视频编程以及使用 INT 33h 对鼠标进行编程。

INT 16h 在读取键盘扩展键（如功能键和光标方向键等）时非常有用。

键盘硬件和 INT 9h，INT 16h，INT 21h 等一起协同工作，使应用程序能够使用键盘输入。本章中的一个例子演示了如何轮询键盘并在 Esc 键按下时结束循环。

视频显示上的色彩是通过叠加合成三基色来产生的。文本的色彩位映射到了视频显示的属性字节中。

在 BIOS 层次上，有许多有用的控制视频显示的 INT 10h 功能。本章包含了一个卷动彩色窗口并在其中央输出文本的例子。

可使用 INT 10h 来绘制彩色图形。本章中的两个例子演示了如何实现。读者可以使用一个简单的公式把逻辑坐标转换为屏幕坐标（像素位置）。

本章中的一个例子演示了如何通过直接写显存高速地绘制彩色图形。

有很多 INT 33h 的功能可操纵并读取鼠标。本章中的一个例子程序跟踪鼠标的移动以及鼠标按键的单击。

更多信息：深入挖掘有关 BIOS 功能的信息并不容易，因为很多好的参考书籍已经绝版了。下面是作者比较喜欢的一些资料：

- Brown Ralf 和 Jim Kyle 所著的 *PC Interrupts, A Programmer's Reference to BIOS, DOS and Third Party Calls*, Addison-Wesley 出版社，1991 年出版。
- Duncan, Ray 所著的 *IBM ROM BIOS*, Microsoft 出版社，1998 年出版。
- Duncan, Ray 所著的 *Advanced MS-DOS Programming*, 第 2 版，Microsoft 出版社，1988 年出版。
- Gilluwe, Frank van 所著的 *The Undocumented PC: A Programmer's Guide to I/O, CPUs, and Fixed Memory Areas*, Addison-Wesley 出版社，1996 年出版。
- Hogan, Thom 所著的 *Programmer's PC Sourcebook: Reference Tables for IBM PCs and Compatibles*,

Ps/2 Systems, Eisa-Based Systems, MS-DOS Operating System Through Version, Microsoft 出版社, 1991 年出版。

- Kyle, Jim 所著的 *DOS 6 Developer's Guide*, SAMS 出版社, 1993 年出版。
- Muhammad Ali, Mazidi 和 Janice Gillispie Mazidi 所著的 *The 80x86 IBM PC & Compatible Computers*, 第 4 版, 卷 I 和卷 II, Prentice-Hall 出版社, 2002 年出版。

本书的 Web 站点 (www.asmirvine.com) 上有很多资源的链接, 包括 Ralf Brown 的 MS-DOS 和 BIOS 中断列表。

15.8 练习题

下面的练习必须在实地址模式下完成。

1. ASCII 码表

使用 INT 10h 显示 IBM 扩展 ASCII 码字符集 (在本书前言部分) 的所有 256 个字符。每行显示 32 个字符, 每个字符后面跟一个空格。

2. 卷动文本窗口

定义一个文本窗口, 大约占视频显示的 3/4 左右, 让程序按顺序执行下面的动作:

- 在窗口的顶行绘制一个随机字符串 (可调用 Irvine16 库中的 Random_range 过程)。
- 窗口向下卷动一行。
- 暂停程序大约 200 ms (可调用 Irvine16 库中的 Delay 函数)。
- 绘制另外一行随机文本。
- 持续卷动和绘制直到显示完 50 行文本为止。

3. 卷动彩色列

以卷动文本窗口的练习 (第 2 题) 作为起点, 做以下更改:

- 随机字符串中的字符只能在列 0, 3, 6, 9, ..., 78 中出现, 其他列应当为空白, 这将创造出列在向下滚动的效果。
- 每列使用一种不同的颜色。

4. 以不同的方向卷动列

以卷动文本窗口的练习 (第 2 题) 为起点, 做以下更改: 在循环开始之前, 随机选择每列的滚动方向是向上还是向下的, 每列的滚动方向在程序运行期间应保持不变。提示: 把每列定义为一个独立的滚动窗口。

5. 使用 INT 10h 绘制矩形

使用 INT 10h 绘制像素的能力创建一个过程 DrawRectangle 来绘制矩形, 接收的参数是矩形左上角和右下角的位置以及绘制的颜色。写一个小测试程序, 使用 INVOKE 指令绘制几个不同大小和颜色的矩形。

6. 使用 INT 10h 绘制函数曲线

使用 INT 10h 绘制像素的能力, 绘制由等式 $Y = 2(X^2)$ 确定的函数曲线。

7. 在模式 13h 中绘制垂直线

修改 15.5.2 节的内存映射图形程序，以便绘制一条垂直线。

8. 在模式 13h 中绘制多条垂直线

修改 15.5.2 节的内存映射图形程序，以便能绘制多条垂直线，每条线使用一种不同的颜色。

9. 边框绘制程序

20 世纪 80 年代和 90 年代早期的 MS-DOS 应用程序在文本模式下通常使用画线字符显示方块和框架。本练习的目的是重现这些编程技术。写一个过程，在屏幕的任何位置绘制单线边框，请使用来自本书前言部分的下列扩展 ASCII 码：C0h, BFh, B3h, C4h, D9h 和 DAh。过程的唯一参数应该是一个指向 FRAME 结构的指针：

```
FRAME STRUCT
    Left BYTE ?           ; 左侧边
    Top  BYTE ?           ; 顶线
    Right BYTE ?          ; 右侧边
    Bottom BYTE ?         ; 底线
    FrameColor BYTE ?     ; 框架颜色
FRAME ENDS
```

写一个程序，向该过程传递各种 FRAME 对象进行测试。

第 16 章 高级 MS-DOS 程序设计

本章要点

- 简介
- 定义段
- 程序的运行时结构
- 中断处理
- 使用 I/O 端口控制硬件

16.1 简介

如果读者想成为在 Intel 处理器硬件层次上编程的工程师，或者想知道若干年前 MS-DOS 专家们如何利用极为有限的资源编写出那么多惊人的作品，那么本章将是很好的阅读材料。如果读者想成为系统级的程序员，本章也将提供一些背景知识。本章主要讲述 MS-DOS 的系统资源以及 MS-DOS 程序设计，主要内容涉及以下方面：

- 如何使用 .MODEL, .CODE, .STACK 及相关伪指令来获得最大的灵活性。
- 如何使用显式段定义伪指令来定义段。
- 示范有多个代码段和数据段的大内存模式程序。
- 解释 COM 和 EXE 程序的运行时结构以及 EXE 文件头。
- 解释程序段前缀 (PSP) 并讲述如何找到 MS-DOS 环境字符串。
- 通过编写一个 Ctrl-Break 中断处理程序（也称中断服务例程，Interrupt Service Routine，简称 ISR），讲述如何用自己编写的中断处理程序替换现存的中断处理程序。
- 解释硬件中断是如何工作的，并且列出了 8259 可编程中断控制器（PIC）使用的各种中断请求优先级（IRQ level）。
- 编写一个截取 Ctrl-Alt-Del 键的常驻内存程序（TSR，Terminate and Stay Resident，终止并驻留）。如果学会了这项技术，读者就可以加入 MS-DOS 专家的行列了。
- 讲述如何向硬件输出端口写数据，如何使用端口监视硬件的状态、控制硬件的行为以及如何从硬件设备读取输入数据。

如果读者跟那些有经验的、从事编程已经有些年头的程序员在一起呆过一阵，那么很可能听到过前面列表中的很多术语。读者是否注意过，那些 DOS 时代的老专家们似乎总是在谈话中不经意地用到 IRQ, TSR, PSP 和 8259 等名词？现在读者可以理解他们以前谈论的究竟是什么了。

16.2 定义段

使用老版本的 MASM 编译的程序必须相当详细地定义代码段、数据段和堆栈段的各种细节。当简化段定义伪指令（.code, .stack, .data）出现的时候，教师们无不如释重负，因为这使他们第一周的课程讲述起来更加容易。显然，专业程序员或许喜欢灵活胜过简化，并仍然坚持用传统的方式编写代码。如果你已经读到了本章（并且理解了前面的所有章节），现在就做好准备去学习显式段定义伪指令晦涩的细节吧。

不过,我们首先还是要先研究一下简化段定义伪指令的各种使用方式。在某些情况下,简化段定义可能就已经能满足需要了。

16.2.1 简化段定义伪指令

使用.MODEL 伪指令的时候,汇编器为近程(NEAR)数据段自动定义了DGROUP段组,DGROUP段组内的段存放近程数据,它可以直接通过DS或SS访问。

.DATA和.DATA?伪指令也创建近程数据段,在运行于实地址模式下时每个段的最大长度是64 Kb。.DATA和.DATA?伪指令创建的数据段放在一个特别的DGROUP段组中,DGROUP的最大长度也是64 Kb。在小型和中型内存模式下使用.FARDATA和.FARDATA?时,汇编器相应创建名为FAR_DATA和FAR_BSS的数据段。

识别变量所在的段:一些BIOS和DOS功能调用在传递参数时要求使用特定的段寄存器。可使用SEG操作符把段的地址赋给段寄存器,例如下面的代码把DS设置为包含farvar的段:

```
mov ax,SEG farvar
mov ds,ax
```

代码段:代码段是用.CODE伪指令定义的。在小内存模式的程序中,.CODE伪指令使汇编器生成一个名为_TEXT的段,在列表文件的段和段组小节中可以看到这一点:

```
_TEXT . . . . .16 Bit 0009      Word Public 'CODE'
```

(这表示是一个名为_TEXT的16位段,长度为9字节长,以偶数字地址边界对齐,并且是一个公共段,段的类名称是'CODE')。

在中型、大型和巨型模式的程序中,每个源代码模块都被赋予了一个不同的段名字,该段名是由模块名加后缀_TEXT构成的。例如汇编器为使用.MODEL LARGE伪指令的MyProg.asm程序生成的列表文件中,代码段的定义如下所示:

```
MYPROG_TEXT . . . . .16 Bit 0009      Word Public 'CODE'
```

不管内存模式如何,都可以在同一个模块中声明多个代码段,声明的方法是在.CODE伪指令后加上可选的段名:

```
.code MyCode
```

应该记住一点:如果代码调用了本书的16位链接库中例程,那么这些代码只能在名为_TEXT的段中。例如,下面的语句将导致链接器产生“fixup overflow”消息:

```
.code MyCode
    mov dx,OFFSET msg
    call WriteString
```

有多个代码段的程序:下面的MultCode.asm程序包含两个代码段,程序中没有使用包含文件Irvine16.inc,以便读者能看清程序中使用的所有MASM伪指令:

```
TITLE Multiple Code Segments      (MultCode.asm)

; This small model program contains multiple
; code segments.

.model small,stdcall
.stack 100h
WriteString PROTO

.data
```

```

msg1 db "First Message",0dh,0ah,0
msg2 db "Second Message",0dh,0ah,"$"

.code
main PROC
    mov     ax,@data
    mov     ds,ax

    mov     dx,OFFSET msg1
    call    WriteString        ; 近调用
    call    Display            ; 远调用
    .exit
main ENDP

.code OtherCode
Display PROC FAR
    mov     ah,9
    mov     dx,offset msg2
    int     21h
    ret
Display ENDP
END main

```

在前面的例子中，_TEXT 段包含 main 过程，OtherCode 段包含 Display 过程，注意 Display 过程必须有一个 FAR 修饰符，以通知汇编器生成远调用（FAR）指令，远调用指令会在堆栈上保存段地址和偏移地址。为了确认声明有效，我们可以在 MultCode.lst 列表文件中看到两个代码段的名字：

```

OtherCode . . . .16 Bit 0008      Word Public 'CODE'
_TEXT . . . . .16 Bit 0014      Word Public 'CODE'

```

16.2.2 显式段定义

有些时候可能需要显式地定义段。例如，或许读者需要在多个数据段内定义额外的内存缓冲区，或许需要把自己的程序同使用私有段名字的目标库相链接。再比如，读者可能正在编写一段要从高级语言中调用的过程，而高级语言编译器并不使用 Microsoft 的段命名约定。

使用显式段定义的程序要完成两个任务：首先，在使用每个段寄存器（DS，ES 和 SS）之前必须设置为相应的段地址；其次，必须告诉汇编器如何正确计算相应段内标号的偏移。

SEGMENT 和 ENDS 定义段的开始和结束。程序中可包含任意数目的段，但每个段都必须有唯一的名字，段还可以组合在一起。段定义的格式如下所示：

```

段名 SEGMENT [对齐方式] [组合方式] ['类']
    语句
段名 ENDS

```

- 段名——标识段的名称，它可以是唯一的名称或是已存在的段名。
- 对齐方式——可以是 BYTE，WORD，DWORD，PARA 和 PAGE。
- 组合方式——可以是 PRIVATE，PUBLIC，STACK，COMMON，MEMORY 或“AT 地址”。
- 类——是用单引号括起的用于标识特定类型段的名称，如 CODE 和 STACK 等。

例如，下面语句定义了 ExtraData 段：

```

ExtraData SEGMENT PARA PUBLIC 'DATA'
    var1 BYTE 1
    var2 WORD 2
ExtraData ENDS

```


对齐方式

在组合两个或多个段时, 对齐方式通知汇编器如何对齐 (定位) 段的起始地址, 默认的对齐方式是 **PARA**, 表示段从以 16 字节对齐的边界开始。例如, 下面都是以 16 字节边界对齐 (以十六进制格式表示) 的 20 位的段地址, 注意最后一个十六进制数字位总是 0:

0A150 81B30 07460

为了创建特定的段对齐, 汇编器在需要时会在现存段的后面插入一些字节, 直到到达正确的起始地址为止。这些额外的字节称为填充字节。对齐方式只影响那些要连接在现存段后面的段, 因为每个段组中的第一个段总是从节边界对齐的地址上开始的 (回忆一下, 第 2 章中曾经讲过段地址的低 4 位总是隐含为 0 的)。程序中使用下列对齐方式:

- **BYTE** 对齐方式使段从前面段结尾处的下一个字节开始。
- **WORD** 对齐方式使段从下一个字对齐的边界地址开始。
- **DWORD** 对齐方式使段从下一个双字对齐的边界地址开始。
- **PARA** 对齐方式使段从下一个 16 字节对齐的边界地址开始。
- **PAGE** 对齐方式使段从下一个 256 字节对齐的边界地址开始。

如果程序要在 8086 和 80286 处理器上运行, 那么数据段使用 **WORD** 对齐方式是最好的, 因为处理器的数据总线是 16 位的, 每次都要读取两个字节, 并且其中第一个字节的地址必须是偶数, 因此存取偶数字节地址开始的变量只需要一次内存读操作, 而对于奇数字节地址开始的变量就需要两次内存读操作。IA-32 体系结构的处理器每次从内存中读取 32 位 (4 字节), 因此应该使用 **DWORD** 对齐方式。

组合方式

段的组合方式通知链接器以何种方式组合有相同名字的段。默认情况下的组合方式是 **PRIVATE**, 表示段不和其他任何段进行组合。

PUBLIC 和 **MEMORY** 组合方式使链接器把该段和其他所有具有相同组合类型的同名段进行组合, 事实上, 组合后所有的段将成为一个单独的段, 所有标号的偏移地址也要做相应的调整, 以便使它们全都从组合后新段的起始地址开始计算。

STACK 组合方式同 **PUBLIC** 类型非常相似, 所有具有 **STACK** 类型的段都要组合在一起。在程序运行时, MS-DOS 自动把 **SS** 初始化为组合方式是 **STACK** 的第一个段的段地址, 同时把 **SP** 设置为段的长度减 1。在 **EXE** 程序中, 应该至少有一个 **STACK** 组合方式的段, 否则链接器会显示一条警告信息。

COMMON 组合方式使所有 **COMMON** 类型的同名段从相同的起始地址开始。事实上, 这些段相互覆盖, 所有的变量或标号的偏移都是从同一起始地址开始计算的, 变量之间可能相互覆盖。

“**AT 地址**”组合方式允许创建从某个绝对地址开始的段, 通常用于存取硬件或操作系统预定义位置的数据。段中的变量或数据不能初始化, 但可以创建变量名引用特定的地址, 例如:

```
bios SEGMENT AT 40h
    ORG 17h
    keyboard_flag BYTE ?           ; MS-DOS 键盘标志
bios ENDS

.code
    mov ax,bios                    ; 指向 BIOS 段
    mov ds,ax
    and ds:keyboard_flag,7Fh      ; 清除最高位
```

在上例中,需要使用段超越前缀(DS:),因为 keyboard_flag 不在标准数据段中,段超越前缀将在 16.2.3 节中讲述。

类名称

段的类(class)为段(特别是对那些名字不同的段)提供了另外一种组合方式。段的类是以引号引起的简单字符串(大小写敏感)。不管在原始程序中的位置如何,类名字相同的段总是被放在一起(不是组合成一个段,但在内存中处于相邻位置)。链接器识别标准类 CODE, CODE 类应该用于那些包含指令的段。如果想使用调试器,就必须在程序中包含一个类名为“CODE”的段。

ASSUME 伪指令

ASSUME 伪指令的作用是告诉编译器如何在编译时计算段中变量和标号的偏移地址, ASSUME 伪指令通常直接放在定义代码段的 SEGMENT 伪指令后,其格式要求段寄存器名后跟一个冒号,冒号后跟段的名字:

```
ASSUME segreg : segname
```

ASSUME 并不真正改变段寄存器的值,仍然必须在运行时使用指令把段寄存器设置为正确的段值。代码中可包含任意多的 ASSUME 伪指令,如果编译器遇到一个新的 ASSUME 伪指令,就会从这时开始相应改变计算偏移地址的方式。

下面的 ASSUME 伪指令告诉编译器默认使用 DS 作为 data1 段的段寄存器:

```
ASSUME ds:data1
```

下面的语句把 CS 寄存器同 myCode 联系起来,把 SS 寄存器同 myStack 联系起来:

```
ASSUME cs:myCode, ss:myStack
```

例子:多个数据段

在本节的前面我们看到了一个有两个代码段的程序,现在创建一个包含两个分别名为 data1 和 data2 的数据段的程序,两个段的类型都声明为 DATA。ASSUME 伪指令把 DS 同 data1 相关联,把 ES 同 data2 相关联:

```
ASSUME cs:cseg, ds:data1, es:data2, ss:mystack
data1 SEGMENT 'DATA'
data2 SEGMENT 'DATA'
```

下面是完整的程序清单:

```
TITLE Multiple Data Segments                                (MultData.asm)

; This program shows how to explicitly declare
; multiple data segments.

cseg SEGMENT 'CODE'
    ASSUME cs:cseg, ds:data1, es:data2, ss:mystack

main PROC
    mov ax,data1                ; DS 指向 data1
    mov ds,ax
    mov ax,SEG val2              ; ES 指向 data2
    mov es,ax

    mov ax,val1                  ; ASSUME 已经使用段 data1 了
    mov bx,val2                  ; ASSUME 已经使用段 data2 了
    mov ax,4C00h                 ; 退出程序
```

```

    int 21h
main ENDP
cseg  ENDS

data1 SEGMENT 'DATA'
    val1 WORD 1001h
data1 ENDS

data2 SEGMENT 'DATA'
    val2 WORD 1002h
data2 ENDS

mystack SEGMENT para STACK 'STACK'
    BYTE 100h DUP('S')
mystack ENDS
END main

```

上面的代码使用了两种方法设置寄存器的值,第一种方法是使用一个段名(data1):

```

mov  ax,data1          ; DS 指向 data1
mov  ds,ax

```

第二种方法是使用 SEG 操作符获取 val2 所在段的段地址:

```

mov  ax,SEG val2        ; ES 指向 data2
mov  es,ax

```

汇编器创建的列表文件表明两个变量 val1 和 val2 的偏移值是相同的,但段属性不同:

名称	类型	值	属性
val1	Word	0000	data1
val2	Word	0000	data2

16.2.3 段超越前缀

段超越前缀是指令前一个字节的前缀,指示处理器在计算有效地址时使用不同于 ASSUME 伪指令指定的另外一个不同的段寄存器。例如,可以使用段超越前缀访问其他段内的变量,而不是当前以 DS 为默认寄存器的段内的变量:

```

mov  al,cs:var1         ; 由 CS 指向的段
mov  al,es:var2         ; 由 ES 指向的段

```

应该注意,在实地址模式下可以把变量放在代码段中,在保护模式程序中不要这样做!

下面的指令获取不由 DS 或 ES 指向的某个其他段内的变量的偏移地址:

```

mov  bx,OFFSET AltSeg:var2

```

如果要多次引用另外一个段内的变量,那么应该插入 ASSUME 语句来改变默认的段引用方式:

```

ASSUME ds:AltSeg          ; 暂时使用段 AltSeg
mov  ax,AltSeg
mov  ds,ax
mov  al,var1
.
.
ASSUME ds:data            ; 使用默认的段 data
mov  ax,data
mov  ds,ax

```

16.2.4 段的组合

本书前面多处提到大型程序应分成多个独立的模块,以简化编辑和调试的过程。即使是不同

模块内的源代码，也可以组合到同一个段中，只要对不同模块内的段使用同样的名字并全部指定 PUBLIC 组合方式即可，这正是在把使用简化段定义伪指令的 16 位程序同本书的 Irvine16 链接库相链接的时候，后台真正发生的事情。

如果使用 BYTE 对齐方式，每个段都从紧接在前面段结尾处的下一个字节开始。如果使用 WORD 对齐方式，则从下一个以字为边界对齐的地址（偶数地址）开始。默认的对齐方式是 PARA，也就是说每个段从下一个以 16 字节为边界对齐的地址开始。

例子程序：下面看一个例子，它有两个程序模块，共有两个代码段、两个数据段和一个堆栈段，最后要组合成三个段（CSEG，DSEG 和 SSEG）。主模块包含三个段，CSEG 和 DSEG 的组合方式都是 PUBLIC，CSEG 段使用了 BYTE 对齐方式以避免在两个代码段连接的时候出现空隙。

主模块

```
TITLE Segment Example          (main module, Seg2.asm)

EXTRN var2:WORD, subroutine_1:PROC

cseg SEGMENT BYTE PUBLIC 'CODE'
ASSUME cs:cseg,ds:dseg,ss:sseg

main PROC
    mov     ax,dseg              ; 初始化 DS
    mov     ds,ax

    mov     ax,var1              ; 局部变量
    mov     bx,var2              ; 外部变量
    call    subroutine_1         ; 外部过程

    mov     ax,4C00h             ; 退出返回 OS
    int     21h

main ENDP
cseg ENDS

dseg SEGMENT WORD PUBLIC 'DATA' ; 局部数据段
    var1 WORD 1000h
dseg ends

sseg SEGMENT STACK 'STACK'      ; 堆栈段
    BYTE 100h dup('S')
sseg ENDS

END main
```

子模块

```
TITLE Segment Example          (submodule, Seg2a.ASM)

PUBLIC subroutine_1, var2

cseg SEGMENT BYTE PUBLIC 'CODE'
ASSUME cs:cseg, ds:dseg

subroutine_1 PROC                ; 从 MAIN 中调用
    mov     ah,9
    mov     dx,OFFSET msg
    int     21h
    ret
subroutine_1 ENDP
cseg ENDS
```

```
dseg SEGMENT WORD PUBLIC 'DATA'

var2 WORD 2000h           ; 由 MAIN 访问
msg  BYTE 'Now in Subroutine_1'
      BYTE 0Dh, 0Ah, '$'

dseg ENDS
END
```

从链接器创建的 MAP 文件可以看到, 程序有一个代码段、一个数据段和一个堆栈段:

Start	Stop	Length	Name	Class
00000H	0001BH	0001CH	CSEG	CODE
0001CH	00035H	0001AH	DSEG	DATA
00040H	0013FH	00100H	SSEG	STACK

Program entry point at 0000:0000

16.2.5 本节习题

1. SEGMENT 和 ENDS 伪指令的作用是什么?
2. SEG 操作符返回的值是什么?
3. 解释一下 ASSUME 伪指令的功能。
4. 在段定义中有哪些可用的对齐方式?
5. 在段定义中有哪些可用的组合方式?
6. 对 IA-32 处理器而言, 哪种对齐方式是最有效率的?
7. 段定义中组合方式的作用是什么?
8. 如何定义一个从绝对地址 (例如 40h) 开始的段?
9. 段定义中类名称选项的作用是什么?
10. 写一条使用段超越前缀的指令。
11. 在下面的例子中, 假设 segA (第一个段) 从地址 1A060h 开始, 那么第三个段 (名字也是 segA) 的起始地址是什么?

```
segA SEGMENT COMMON
    var1 WORD ?
    var2 BYTE ?
segA ends

stack SEGMENT STACK
    BYTE 100h dup(0)
stack ends

segA SEGMENT COMMON
    var3 WORD 3000h
    var4 BYTE 40h
segA ends
```

16.3 程序的运行时结构

熟练的汇编语言程序员需要了解大量的 MS-DOS 相关知识。本节讲述 command.com、程序段前缀、COM 及 EXE 程序的结构。

MS-DOS 和 Windows 95/98 自带的 command.com 程序称为命令行处理器, 在 Windows 2000/XP 下没有 command.com, 命令行处理器是 cmd.exe。命令行处理器解释用户在命令行提示符下输入的命令。输入一条命令时, 系统会产生下面的动作序列:

1. MS-DOS 检查命令是否属于内部命令（例如 DIR、REN 或 DEL 等），内部命令由驻留内存的 MS-DOS 的过程立即执行。
2. MS-DOS 查找是否有与输入命令匹配的以 COM 为扩展名的文件名，如果有匹配文件并且在当前目录中，该文件马上被执行。
3. MS-DOS 查找是否有与输入命令匹配的以 EXE 为扩展名的文件，如果有匹配文件并且在当前目录中，该文件马上被执行。
4. MS-DOS 查找是否有与输入命令匹配的以 BAT 为扩展名的匹配文件，如果有匹配文件并且在当前目录中，该文件会马上被执行。以 BAT 为扩展名的文件称为批处理文件，批处理文件实际上是一个文本文件，其中包含的是要执行的 MS-DOS 命令，执行批处理文件中的命令和在终端键入命令执行是一样的，不同的是批处理中可以一次执行多条 MS-DOS 命令，而在终端上每次只能键入执行一条 MS-DOS 命令。
5. 如果 MS-DOS 在当前目录中找不到匹配的 COM、EXE 或 BAT 文件，则查找当前路径中的第一个目录，如果还是找不到就继续查找第二个目录，MS-DOS 循环执行此过程，直到找到一个匹配的文件或路径中所有的目录都搜索完毕为止。

以 COM 和 EXE 为扩展名的程序称为暂留程序（相对于驻留程序而言），它通常被装入到足够容纳程序的内存段中执行。当执行完毕后，占用的内存随之释放。如有必要，暂留程序可以在退出时把一部分代码保留在内存中，这样的程序称为内存驻留程序（TSR）。

16.3.1 程序段前缀

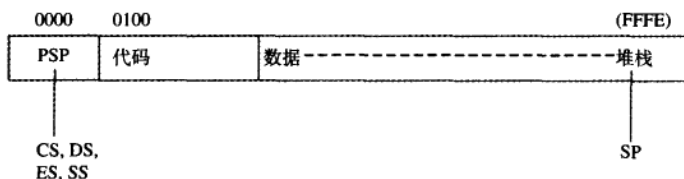
在程序加载到内存中时，MS-DOS 在程序的开始为其创建一个特殊的 256 字节的信息块，称为程序段前缀（PSP, Program Segment Prefix）。程序段前缀的结构如表 16.1 所示。

表 16.1 程序段前缀(PSP)

偏移地址	内 容
00~15	一些 MS-DOS 指针和中断向量地址
16~2B	MS-DOS 保留
2C~2D	当前环境字符串的段地址
2E~5B	MS-DOS 保留
5C~7F	文件控制块 (FCB) 1 和 2，主要由 MS-DOS 2.0 以前的程序使用
80~FF	用于存放 MS-DOS 命令行的一份副本，同时也作为默认的磁盘传输缓冲区

16.3.2 COM 程序

暂留程序有两种，这取决于程序文件使用的扩展名（COM 或 EXE）。COM 程序是未经修改的机器语言程序的二进制映像。MS-DOS 会把 COM 程序加载到最低可用的段地址，并在偏移 0 处创建一个 256 字节的 PSP，代码、数据和堆栈都位于同一个物理段（或逻辑段）中。COM 程序的最大长度可以到 64 KB，其中代码占用的空间最大不能超过 64 KB 减去 PSP 的大小（256 字节），以及为堆栈底端保留的两个字节。如下图所示，所有的段寄存器都被设置为指向 PSP 的基地址，代码从偏移 100h 处开始，数据区紧接在代码区之后，堆栈区在段的最底端，MS-DOS 将 SP 初始化为 FFFh：



下面来看一个简单的 COM 格式的程序。MASM 要求 COM 程序使用微型内存模式，还必须使用 ORG 伪指令把程序代码的起始地址设为 100h，这就为 PSP 保留了从 0 到 0FFh 之间的 100h 字节可用空间：

```
TITLE Hello Program in COM format    (HelloCom.asm)
.model tiny
.code
org 100h                               ; 必须放在 main 之前
main PROC
    mov     ah,9
    mov     dx,OFFSET hello_message
    int     21h
    mov     ax,4C00h
    int     21h
main ENDP

hello_message BYTE 'Hello, world!',0dh,0ah,'$'
END main
```

因为没有单独的段用于存放数据，所以通常把变量放在主过程后面。如果把数据放在程序的开始，CPU 就会试图去执行数据。一个变通的方法是在程序的开始使用一条 JMP 指令跳过数据区，跳到第一条实际的指令处去执行：

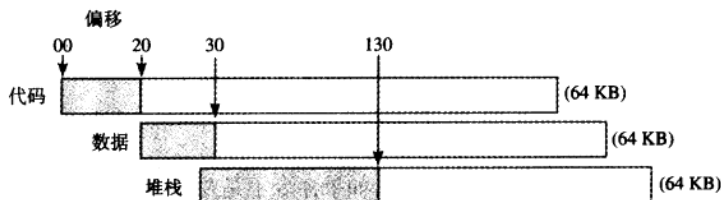
```
TITLE Hello Program in COM format    (HelloCom.asm)
.model tiny
.code
org 100h                               ; 必须放在入口点之前
main proc
    jmp     start                       ; 跳过数据
hello_message BYTE 'Hello, world!',0dh,0ah,'$'
start:
    mov     ah,9
    mov     dx,OFFSET hello_message
    int     21h
    mov     ax,4C00h
    int     21h
main ENDP
END main
```

Microsoft 的链接器要求使用 /T 参数，以便通知它创建一个 COM 文件而不是 EXE 文件。对于同样源码编译的程序，COM 格式的程序总是比 EXE 格式的程序要小。例如，当保存在磁盘上的时候，HelloCom.asm 的 COM 格式程序只有 17 字节长，但是当装入内存后，COM 程序总是要占用 64 KB 大小的内存，而不管它实际是否需要这么多的空间。COM 程序格式不是为运行于多任务环境下而设计的。

16.3.3 EXE 程序

磁盘上的 EXE 程序由 EXE 文件头和紧接其后的可装载程序模块构成，后面的可装载程序模块才是程序本身。程序头部并不装入内存，相反，它所包含的只是一些 MS-DOS 加载和执行程序时需要的信息。

MS-DOS 加载 EXE 程序的时候在最低的可用地址处创建程序段前缀 (PSP)，程序则置于 PSP 之上。MS-DOS 把 DS 和 ES 设置为指向程序的加载地址（程序段前缀 PSP 的起始地址）；CS 和 IP 设置为指向代码的入口，程序将从入口地址开始执行；SS 被设置为指向堆栈段的开始，SP 则设为堆栈段的大小。下面是存在交叉的代码段、数据段和堆栈段的示意图：



程序中代码区占用 20h 字节，数据区占用 10h 字节，堆栈区占用 100h 字节。

EXE 程序最多可以包含 65535 个段，尽管正常情况下不会用到那么多。如果程序有多个数据段，那么程序员必须手动设置 DS 和 ES，以访问每个数据段。

内存使用

EXE 使用的内存数量是由程序头决定的——特别是程序代码段之上需要的最少和最多的节数（每节等于 16 字节），这部分内存用于在程序运行时存放变量以及堆栈数据。默认情况下，链接器把最大值设为 65535 个节，这通常超过了一般情况下 MS-DOS 中最多可获得的内存数量，因此在程序被加载的时候，MS-DOS 将自动分配所有可用的内存。

链接时可使用 /CP 选项来设置最多分配的内存数量。以下面的 prog1.obj 为例，其中 1024 指以 16 字节为单位的节的数目（十进制）：

```
link16 /cp:1024 prog1;
```

也可以使用 Microsoft 汇编器附带的 exehdr 程序在编译链接后再修改这些值，例如下面的命令行可以把 prog1.exe 程序的最大内存分配值设为 400h 个节（16384 字节）：

```
exehdr prog1 /max 400
```

exehdr 还可以显示一些重要的程序统计信息，prog1.exe 程序在以最大内存分配值 1024 链接后的输出如下所示：

PROG1	(Hex)	(Dec)
EXE size (bytes)	876	2166
Minimum load size (bytes)	786	1926
Overlay number	0	0
Initial CS:IP	0000:0010	16
Initial SS:SP	0068:0100	256
Minimum allocation (para)	11	17
Maximum allocation (para)	400	1024
Header size (para)	20	32
Relocation table offset	1E	30
Relocation entries	1	1

EXE 文件头

MS-DOS 使用 EXE 文件头中的信息来计算段以及一些其他组件的正确地址。文件头中包含了以下信息:

- 重定位表, 包含程序加载时要重新计算并加以修正的地址。
- EXE 程序的大小, 以 512 字节为单位。
- 最少内存分配数量: 在程序映像之上需要的最少内存数量 (按节计算), 这部分内存可能会用于运行时存放的动态数据的堆。
- 最多内存分配数量: 在程序映像之上需要的最多内存数量 (按节计算)。
- IP 和 SP 寄存器的初始值。
- 堆栈和代码段距加载模块开始的偏移 (以 16 字节的节为单位)。
- 文件的校验和。向内存中装入程序时用于捕捉数据错误。

16.3.4 本节习题

1. 在 MS-DOS 命令提示符下键入命令时, 如果命令不是 MS-DOS 内部命令, 将会发生什么?
2. 执行命令时, MS-DOS 在当前目录中查找 EXE 文件之前优先查找 BAT 文件吗?
3. 什么是暂留程序?
4. 暂留程序开始的 256 字节区域称为什么?
5. 暂留程序在何处保存当前环境字符串的段地址?
6. COM 程序是什么?
7. COM 程序使用的内存模式是什么?
8. 使用链接器创建 COM 程序时需要指定什么命令行选项?
9. 使用 COM 程序有什么内存限制?
10. 在运行时, COM 程序对内存使用的效率高吗?
11. COM 程序中可以包含多少个段?
12. COM 程序的段寄存器初始值分别是什么?
13. ORG 伪指令的作用是什么?
14. 存储在磁盘上的 EXE 程序由文件头和_____模块两个主要的部分构成。
15. EXE 程序加载后 DS 和 ES 寄存器指向哪里?
16. 分配给 EXE 程序的内存数量由什么来决定?
17. exehdr 程序有哪些功能?
18. 如果想知道 EXE 文件中的文件重定位表项的数目, 应该到何处查找?

16.4 中断处理

本节讨论通过安装中断处理程序 (中断服务例程) 定制 BIOS 和 MS-DOS 的方法。在前面的章节中已经讲过, BIOS 和 MS-DOS 包含了很多能够简化输入/输出和基本系统任务的中断处理例程。到目前为止, 我们已经看到了很多实例——用于视频操作的 INT 10h, INT 16 键盘服务以及 INT 21h 磁盘服务等。操作系统中同样重要的一部分是那些响应硬件中断的处理例程。MS-DOS 允许用户自己编写的中断服务例程替换任何系统的服务例程。

限制: 本章中讲述的中断处理过程只有在引导到 MS-DOS 模式下才能正常工作。读者可

以在 Windows 95/98 下的 MS-DOS 方式中运行这些中断服务例程,但在 Windows NT/2000/XP 下不能正常工作。后者屏蔽了应用程序对系统硬件的访问,以获得更好的稳定性和安全性。如果操作系统允许同时运行的两个程序修改同一硬件设备的内部配置,那么结果可能是无法预料的。

编写中断服务过程的理由可能有很多。读者或许想在热键按下时激活自己的程序,即使这时用户正在运行其他的应用程序。Borland 的 SideKick 是第一个能够无论在何时(只要用户按下特定的热键组合)都能够弹出一个记事本或计算器的程序。

读者可以用自己的中断服务例程替换掉 MS-DOS 默认的中断服务例程,以提供更完善的服务。例如,当 CPU 试图以 0 去除一个数字的时候就会触发一个除 0 错误中断,但是系统没有任何标准的方式使程序恢复执行。

读者还可以用自己的中断处理例程替换掉 MS-DOS 的严重错误处理例程或 Ctrl-Break 处理例程。MS-DOS 默认的严重错误处理例程终止程序并返回 MS-DOS,但是使用读者自己的错误处理例程就可以从错误中恢复并允许用户继续运行当前的应用程序。

用户自己编写的中断处理例程可以比 MS-DOS 更有效地处理硬件中断。例如,PC 异步通信处理例程对输入/输出不进行任何缓冲,这就意味着如果输入字符在另外一个字符到达之前没有被及时地从端口输入的话就会丢失。内存驻留程序可以等待输入字符产生硬件中断,然后从端口输入字符并把字符存储在一个缓冲区中,这样应用程序就不需要不断地检查串行端口了,从而节省出大量的宝贵时间。

中断向量表: MS-DOS 如此灵活的秘密在于内存开始 1024 字节的中断向量表中(从 0:0 开始到 0:03FF)。表 16.2 包含一些中断向量表的例子,表中的每个项(称为中断向量)是一个 32 位的段-偏移形式的地址,它指向已存在的中断服务例程。

表 16.2 中断向量表的例子

中断向量号	偏移	中断向量			
00~03	0000	02C1:5186	0070:0C67	0DAD:2C1B	0070:0C67
04~07	0010	0070:0C67	F000:FF54	F000:837B	F000:837B
08~0B	0020	0D70:022C	0DAD:2BAD	0070:0325	0070:039F
0C~0F	0030	0070:0419	0070:0493	0070:050D	0070:0C67
10~13	0040	C000:0CD7	F000:F84D	F000:F841	0070:237D

由于 BIOS 和 MS-DOS 的版本可能不同,因此在不同的计算机上,这些中断向量的值可能也会有所不同。在表中,INT 0 的中断处理过程(除 0 错)的地址是 02C1:5186h,任何中断向量的偏移地址都可以通过中断向量号乘以 4 找到,INT 9h 中断向量的偏移地址就是 $9 * 4$,即十六进制的 0024。

中断处理过程的执行: 中断处理过程在下面两种情况下将被执行:(1)包含 INT 指令的应用程序在执行 INT 指令时会自动调用中断处理过程,这就是所谓的软件中断;(2)另一种方法是通过硬件中断,硬件中断在硬件设备(异步端口、键盘和定时器等)向可编程中断控制芯片发送信号时发生。

16.4.1 硬件中断

硬件中断是由 Intel 8259 可编程中断控制器(PIC)产生的,PIC 通知 CPU 挂起当前正在执行

的程序并执行中断服务例程。如果没有中断服务例程把等待在键盘输入端口的字符或串口收到的字符保存在缓冲区中,那么这些字符就可能会丢失。

有时,程序在对段寄存器或堆栈进行某些敏感操作的时候必须禁止硬件中断。CLI (clear interrupt flag) 指令可以禁止中断,STI (set interrupt flag) 指令能够允许中断的发生。

中断请求优先级 (IRQ Level): 中断可以由 PC 上的许多设备触发,其中的部分列在表 16.3 中。每个设备都有一个基本的中断请求 (IRQ) 优先级。表 16.3 中优先级 0 的级别最高,优先级 15 的级别最低。较低优先级的中断不能中断较高优先级的中断处理执行过程。例如,如果串口 1 (COM1) 试图中断键盘中断处理例程时,就不得不等待直到前者的处理过程完成。同时发出的两个或多个中断请求依据中断优先级的高低依次被处理。8259 PIC 负责中断的调度。

表 16.3 IRQ 的分配 (ISA 总线)

IRQ	中断号	描 述
0	8	系统定时器 (每秒 18.2 次)
1	9	键盘
2	0Ah	可编程中断控制器 2
3	0Bh	COM2 (串口 2)
4	0Ch	COM1 (串口 1)
5	0Dh	LPT2 (并口 2)
6	0Eh	软盘控制器
7	0Fh	LPT1 (并口 1)
8	70h	CMOS 实时时钟
9	71h	(重定向到 INT 0Ah)
10	72h	(可用) 声卡
11	73h	(可用) SCSI 卡
12	74h	PS/2 鼠标
13	75h	数学协处理器
14	76h	硬盘控制器
15	77h	(可用)

以键盘为例,当键被按下的时候,8259 PIC 向 CPU 发送 INTR 信号并传递中断向量号,如果当前没有禁止外部中断,那么 CPU 按序执行下面的动作:

1. 在堆栈上压入标志寄存器。
2. 清除中断允许标志,禁止任何其他的外部硬件中断。
3. 在堆栈上压入当前的 CS 和 IP。
4. 定位 INT 9 的中断向量表项,然后把它相应的地址装入 CS 和 IP。

接着, BIOS INT 9 服务程序开始执行,并按序执行下面的动作:

1. 重新允许硬件中断,以避免影响系统时钟。
2. 从键盘端口输入一个字符的扫描码并把它转换成 ASCII 码,如果不能转换成 ASCII 码,则把 ASCII 码置为 0,最后把字符的扫描码和 ASCII 码存储在 BIOS 系统数据区中的一个 32 字节的循环使用的缓冲区中。
3. 执行 IRET (中断返回) 指令,从堆栈上弹出 IP, CS 和标志寄存器,这时控制权交还给中断发生时正在执行的程序。

16.4.2 中断控制指令

CPU 有一个中断允许标志 (IF) 控制着 CPU 是否响应外部 (硬件) 中断。如果中断允许标志置位 (IF=1), 则允许外部中断; 如果标志清零 (IF=0), 则禁止外部中断。

STI 指令: STI 指令设置中断允许标志, 使 CPU 能够响应外部中断。例如系统响应键盘输入时要挂起当前正在执行的程序并做以下事情: 调用 INT 9, 该中断在一个缓冲区内存储击键码, 然后返回到当前执行的程序。一般情况下, 中断允许标志是置位的, 否则, 系统定时器将不能正确计算日期和时间, 键盘击键码也会丢失。

CLI 指令: CLI 指令清除中断允许标志, 禁止 CPU 响应外部中断。应该慎用 CLI 指令, 仅应在执行不能被中断的关键操作时使用。例如, 如果代码在修改 SS 和 SP 时被中断, 这时 SS 寄存器可能已经修改了指向一个新的段地址, 但 SP 还没有修改:

```
mov ax,mystack          ; 重设 SS
mov ss,ax
; 在这里中断!!!
mov sp,100h             ; 重设 SP
```

为安全起见, 应使用 CLI 指令禁用中断, 在操作完成后再使用 STI 指令启用中断:

```
cli                      ; 禁用中断
mov ax,mystack           ; 重设 SS
mov ss,ax
mov sp,100h              ; 重设 SP
sti                      ; 重新启用中断
```

每次禁止中断的时间不应该超过几毫秒, 否则就会丢失击键或使系统时钟变慢, CPU 响应软件或硬件中断的时候, CPU 自动禁止其他的外部中断。MS-DOS 和 BIOS 中断处理例程在开始执行时所做的第一件事就是重新允许外部中断。

16.4.3 编写中断处理例程

有人可能会问: 为什么要使用中断向量表? 当然, 我们也可以直接调用 ROM 内的特定过程来处理中断。IBM-PC 设计者使用中断向量表的目的是为了修改和更正 BIOS 处理例程而不必替换 ROM 芯片。通过使用中断向量表, 程序员就可以直接替换中断向量表内的地址, 使其直接指向 RAM 内的处理过程。

中断向量表中的每个表项都指向一个称为中断处理例程或中断服务例程 (ISR) 的过程。应用程序可以用指向新中断处理例程的地址替换中断向量表中原来的地址。例如, 可以用这样的方法写一个自己的键盘中断处理过程。除非万不得已, 否则没有必要费那个劲, 一个可能更好的变通方法是在中断处理过程中直接调用 INT 9 从键盘端口读入一个字符, 按键存入了缓冲区之后再对缓冲区进行操作。

INT 21h 的功能 25h 和 35h 功能使安装中断处理过程成为可能。功能 35h (获取中断向量) 返回段-偏移形式的中断向量地址。调用该功能时 AL 中放入要返回地址的中断向量号, MS-DOS 把中断向量的地址存放在 ES:BX 中返回。例如, 下面的语句返回 INT 9 的中断向量的地址:

```
.data
int9Save LABEL WORD
DWORD ?                ; 这里保存原来的 INT 9 中断地址
```

```
.code
mov ah,35h                ; 获取 INT 9 的中断向量
mov al,9
int 21h                   ; 调用 MS-DOS
mov int9Save,BX           ; 存储偏移地址
mov int9Save+2,ES         ; 存储段地址
```

INT 21h 的功能 25h (设置中断向量) 允许用新的中断处理过程地址替换原中断处理过程。调用时 AL 放入中断向量号, DS:DX 中放入新的段-偏移格式的中断处理例程的地址:

```
mov ax,SEG kybd_rtn        ; 键盘中断处理过程
mov ds,ax                 ; 段
mov dx,OFFSET kybd_rtn     ; 偏移地址
mov ah,25h                ; 设置 INT 9 的中断向量
mov al,9h
int 21h
.
.
kybd_rtn PROC              ; 新的 INT 9 中断处理过程
```

Ctrl-Break 中断处理例程的例子

MS-DOS 程序在等待输入时, 按下 Ctrl-Break 键会使控制转移到 INT 23h 中断处理例程。默认的 Ctrl-Break 中断处理例程结束当前正在运行的程序, 这可能导致当前程序处于一种不稳定的状态, 因为文件可能处于打开状态还未关闭, 或者程序使用的内存尚未释放。不过, 读者可以用自己的代码替换原 INT 23h 中断处理过程, 以阻止 MS-DOS 终结程序。下面的程序安装了一个简单的 Ctrl-Break 处理过程:

```
TITLE Control-Break Handler          (Ctrlbrk.asm)

; This program installs its own Ctrl-Break handler and
; prevents the user from using Ctrl-Break (or Ctrl-C)
; to halt the program. The program inputs and echoes
; keystrokes until the Esc key is pressed.

INCLUDE Irvine16.inc

.data
breakMsg BYTE "BREAK",0
msg BYTE "Ctrl-Break demonstration."
        BYTE 0dh,0ah
        BYTE "This program disables Ctrl-Break (Ctrl-C). Press any"
        BYTE 0dh,0ah
        BYTE "keys to continue, or press ESC to end the program."
        BYTE 0dh,0ah,0

.code
main PROC
    mov ax,@data
    mov ds,ax
    mov dx,OFFSET msg          ; 显示欢迎信息
    call Writestring

install_handler:
    push ds                    ; 保存 DS
    mov ax,@code               ; 初始化 DS 指向代码段
    mov ds,ax
    mov ah,25h                 ; 设置 INT 23h
    mov al,23h                 ; 的中断处理例程
    mov dx,OFFSET break_handler
```

```

        int    21h
        pop    ds                ; 恢复 DS
L1:     mov    ah,1              ; 等待按键并回显
        int    21h
        cmp    al,1Bh           ; 按下了 ESC 键?
        jnz    L1               ; 否: 继续

        exit
main ENDP

; 下面的过程在按下 Ctrl-Break 时执行,
; 必须保护所有的寄存器。
break_handler PROC
        push   ax
        push   dx
        mov    dx,OFFSET breakMsg
        call   WriteString
        pop    dx
        pop    ax
        iret
break_handler ENDP
END main

```

main 过程初始化了 INT 21h 的中断向量。INT 21h 功能 25h 所需的输入参数是:

- AH=25h
- AL= 要处理的中断向量号 (21h)
- DS:DX= 新的 Ctrl-Break 处理程序的段-偏移地址

程序的主循环简单地输入并且回显击键, 直到 Esc 键被按下为止。

在某些系统上, 可能要按 Ctrl-C 键而不是 Ctrl-Break 键去激活 Ctrl-Break 处理过程。

Ctrl-Break 被按下的时候, break_handler 过程被执行, 该过程调用 WriteString 显示一条消息。在 break_handler 结尾处执行 IRET(从中断返回)指令时, 控制权返回到主程序, 这样在 Ctrl-Break 按下时被中断的无论是哪个过程, 都将得以继续执行。通常, 在 Ctrl-Break 中断中可以调用任何 MS-DOS 中断处理例程, 但必须保护所有的寄存器。

程序结束时不必恢复 INT 21h 中断向量, 因为 MS-DOS 在程序结束时自动恢复 INT 21h 中断向量。MS-DOS 把原始的 INT 21h 中断向量存储在程序段前缀 (PSP) 的偏移 000Eh 处。

16.4.4 内存驻留程序

内存驻留 (TSR) 程序一直常驻在内存中, 直到被特殊的工具删除或计算机重启为止。TSR 程序平时处于睡眠状态, 但可以由一些特殊事件 (如按键等事件) 激活。

在使用 TSR 的早期, 两个或多个程序替换同一中断向量时会产生兼容性问题。老一点的程序使中断向量指向自己的程序但并不链接原来替换同一中断向量的例程。后来, 为了补救这类问题, TSR 程序的作者必须保存要替换的中断向量原来的内容, 在自己的中断处理例程完成处理后, 把控制权转移给原来的中断向量处理例程。这种方法较原来方法无疑是一种进步, 但同时也意味着在处理中断时, 后安装的 TSR 自动获得了最高的优先权。用户有时必须非常细心地安排载入 TSR 程序的顺序。在 MS-DOS 流行的时候, 已经出现了管理多个内存驻留程序的商业化工具。

键盘的例子

假设我们要编写一个中断服务例程 (ISR), 可以检查键盘上输入的每个字符并把它存储在位置 10B2:0020 处。要想安装自己的 ISR, 必须首先获取 INT 9 中断向量并保存, 然后用自己的 ISR

地址替换相应的中断向量表项。

一个键被按下的时候, 键盘控制器向计算机的键盘端口传输一个字节, 并引发一个硬件中断。8259 PIC 把中断向量号传递给 CPU, CPU 跳转到中断向量表中 INT 9 处理例程 (也就是我们的 ISR) 所在的地址处执行。我们的处理过程获得了检查键盘输入的机会, 在 ISR 退出时, 执行一条跳转指令跳转到原来的键盘处理过程去继续执行。

这个链起来的过程如图 16.1 所示, 其中的地址是随意假定的。当 BIOS INT 9 处理程序结束时, IRET 指令从堆栈弹出标志寄存器并把控制权交还给按键发生时正在执行的程序。

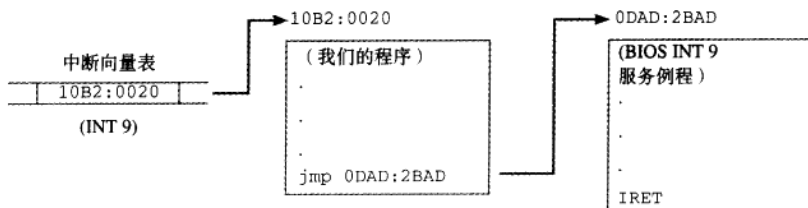


图 16.1 中断向量图示

16.4.5 应用：程序 No_Reset

下面的例子阻止按 Ctrl-Alt-Delete 重启系统, 是一个简单的内存驻留程序。一旦程序被安装到内存中, 系统只能通过按特殊的组合键重启: Ctrl-Alt-右 Shift-Del (卸载该程序的唯一方法是关闭计算机并重新启动)。该程序只能在计算机引导到 MS-DOS 方式下才能正常工作。Microsoft 的 Windows NT/2000/XP 禁止 TSR 程序截取键盘按键。

MS-DOS 键盘状态字节: 在开始之前, 还需要了解一些关于 MS-DOS 存储在低端内存中的键盘状态字节的相关信息, 如图 16.2 所示。程序要检查该标志, 看是否按下了 Ctrl-Alt-右 Shift-Del 键。键盘状态字节存放在内存位置 0040:0017h 处。

另外, 0040:0018h 处的键盘状态字节复制了前面 17h 处的状态字节, 所不同的是它的位 3 表示是否按下了 Ctrl-NumLock 键。

安装程序: 只有在内存中安装了内存驻留代码, 程序才能开始工作。从安装时起, 所有的键盘输入都将由该程序过滤, 如果处理程序有 bug, 那么可能会锁死键盘, 这时只能冷启动机器了。键盘中断处理例程很难调试, 因为在调试程序时要不停地使用键盘。经常写 TSR 程序的专家们通常使用辅助的硬件调试器在受保护的内存中维护一个跟踪缓冲区。通常最难捕捉的 bug 出现在程序实际运行时而不是单步跟踪时。注意: 必须重启计算机到 MS-DOS 模式下再安装本程序。

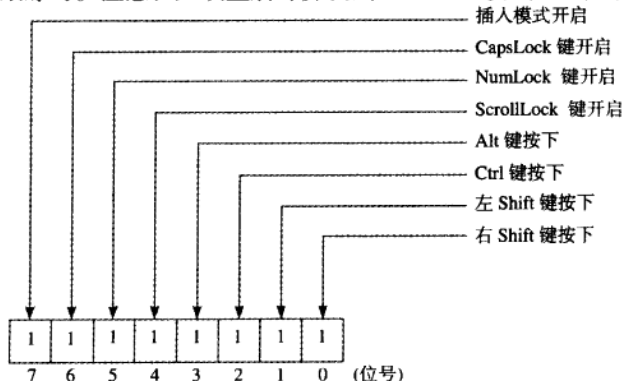


图 16.2 键盘状态标志字节

程序清单：下面的程序清单中，安装部分的代码位于尾部，因为这部分不需要常驻内存。以标号 int9_handler 开始的驻留部分常驻在内存中，int 9h 的中断向量就指向了这部分：

```

TITLE Reset-Disabling program                (No_Reset.asm)

; This program disables the usual DOS reset command
; (Ctrl+Alt+Del), by intercepting the INT 9 keyboard
; hardware interrupt. It checks the shift status bits
; in the MS-DOS keyboard flag and changes any Ctrl+Alt+Del
; to Alt+Del. The computer can only be rebooted by
; typing Ctrl+Alt+Right shift+Del. Assemble, link,
; and convert to a COM program by including the /T
; command on the Microsoft LINK command line.
; Boot into pure MS-DOS mode before running this program.

.model tiny
.386
.code
    rt_shift    EQU 01h                ; 右 Shift 键: 位 0
    ctrl_key    EQU 04h                ; Ctrl 键: 位 2
    alt_key     EQU 08h                ; Alt 键: 位 3
    del_key     EQU 53h                ; Del 键的扫描码
    kybd_port   EQU 60h                ; 键盘输入端口

    ORG 100h                            ; 这是一个 COM 程序
start:
    jmp setup                            ; 跳转到 TSR 安装部分

; 内存驻留代码从这里开始
int9_handler PROC FAR
    sti                                ; 允许硬件中断
    pushf                                ; 保存寄存器和标志
    push es
    push ax
    push di

; ES:DI 指向 DOS 键盘标志字节
L1: mov ax,40h                            ; DOS 数据段从 40h 处开始
    mov es,ax
    mov di,17h                            ; 键盘标志的位置
    mov ah,es:[di]                        ; 复制键盘标志至 AH

; 测试 Ctrl 和 Alt 键
L2: test ah,ctrl_key                      ; 按下了 Ctrl 键?
    jz L5                                ; 否: 退出
    test ah,alt_key                      ; 按下了 Alt 键?
    jz L5                                ; 否: 退出

; 测试 Del 和右 Shift 键
L3: in al,kybd_port                      ; 读取键盘端口
    cmp al,del_key                      ; 按下了 Del 键?
    jne L5                                ; 否: 退出
    test ah,rt_shift                    ; 按下了右 Shift 键?
    jnz L5                                ; 是: 允许系统重启

L4: and ah,NOT ctrl_key                  ; 否: 关闭 Ctrl 位
    mov es:[di],ah                      ; 恢复键盘标志

L5: pop di                                ; 恢复寄存器和标志
    pop ax
    pop es
    popf
    jmp cs:[old_interrupt9]              ; 跳转到原来的 INT 9 处理例程

old_interrupt9 DWORD ?

```



```

int9_handler ENDP
end_ISR label BYTE

; ----- (end of TSR program) -----
; Save a copy of the original INT 9 vector, and set up
; the address of our program as the new vector. Terminate
; this program and leave the int9_handler procedure in memory.

setup:
    mov     ax,3509h                ; 获取 INT 9 中断向量
    int     21h
    mov     word ptr old_interrupt9,bx ; 保存原 INT 9 中断向量
    mov     word ptr old_interrupt9+2,es
    mov     ax,2509h                ; 设置 INT 9 中断向量
    mov     dx,offset int9_handler
    int     21h
    mov     ax,3100h                ; 终止并驻留
    mov     dx,OFFSET end_ISR        ; 指向驻留代码的末尾
    shr     dx,4                    ; 除以 16
    inc     dx                      ; 向上按节近似取整
    int     21h                    ; 执行 MS-DOS 功能调用
END start

```

首先看一下 setup 标号处安装程序的代码, 程序调用 INT 21h 功能 35h 获取当前的 INT 9h 中断向量, 然后保存在变量 old_interrupt9 中, 这样做的目的是为了能够前向链接原来的键盘中断处理过程。接着程序使用 INT 21h 的功能 25h, 把 INT 9h 中断向量的地址指向程序的驻留部分。程序结束时, 调用 INT 21h 的功能 31h 返回 MS-DOS, 并把驻留部分留在内存中, 该功能自动在内存中保留从 PSP 开始到 DX 中指定的偏移处的所有内容。

驻留程序: 内存驻留中断处理例程从标号 int9_handler 处开始, 每次按键的时候该中断处理例程都要被执行。一旦中断处理例程获得控制权, 应该马上重新允许中断, 因为 8259 PIC 自动禁止了中断。

```

int9_handler PROC far
    sti                ; 允许硬件中断
    pushf              ; 保存寄存器和标志
    (等等)

```

要记住的是, 按键动作经常在其他程序执行时发生, 如果在中断处理例程中修改了寄存器和标志的内容, 可能会导致应用程序发生难以预料错误。

下面的语句定位地址 0040:0017h 处的键盘状态字节并复制到 AH 中, 程序要检查该字节, 看一下现在被按下的是什么键:

```

L1:  mov     ax,40h                ; DOS 数据段从 40h 处开始
     mov     es,ax
     mov     di,17h                ; 键盘标志的位置
     mov     ah,es:[di]            ; 复制键盘标志至 AH

```

下面的语句检查是否按下了 Ctrl 和 Alt 键, 如果都没有按下, 则退出:

```

L2:  test    ah,ctrl_key           ; 按下了 Ctrl 键?
     jz      L5                    ; 否: 退出
     test    ah,alt_key            ; 按下了 Alt 键?
     jz      L5                    ; 否: 退出

```

如果 Ctrl 和 Alt 键同时被按下, 表示可能有人正试图重启机器, 为了查看按下的是什么键, 程序从键盘端口输入字符并同 Del 键比较:

```
L3:  in    al,kybd_port      ; 读取键盘端口
     cmp    al,del_key      ; 按下了 Del 键?
     jne    L5              ; 否: 退出
     test   ah,rt_shift     ; 按下了右 Shift 键?
     jnz    L5              ; 是: 允许系统重启
```

如果用户并没有按下 Del 键, 程序就简单地退出并且让 INT 9h 来处理击键。如果 Del 键被按下, 就知道有人正按下 Ctrl-Alt-Del 重启机器。只有在同时按下右 Shift 键的时候才允许用户重启机器, 否则程序就清除键盘标志字节的 Ctrl 位, 这样就有效地阻止了用户试图重启机器的企图:

```
L4:  and    ah,NOT ctrl_key  ; 否: 关闭 Ctrl 位
     mov    es:[di],ah      ; 恢复键盘标志
```

最后程序执行一条远跳转指令跳到原 BIOS INT 9h 中断处理例程, 该中断处理例程地址存储在 old_interrupt9 变量中, 于是计算机就可以处理正常的击键了, 这对于计算机的基本操作是非常重要的:

```
jmp cs:[old_interrupt9]      ; 跳转到原来的 INT 9 处理例程
```

16.4.6 本节习题

1. 严重错误处理例程的默认动作是什么?
2. 中断向量表每个项所包含的内容是什么?
3. INT 10h 中断向量存储在哪个地址中?
4. 哪个控制器芯片产生硬件中断?
5. 哪条指令禁止硬件中断?
6. 哪条指令允许硬件中断?
7. IRQ 0 和 IRQ 15 哪个的优先级最高?
8. 如果一个程序正在创建一个磁盘文件, 这时按下一个键, 根据你对中断请求优先级的了解, 你认为是在文件创建之前还是之后按键码被送入键盘缓冲区?
9. 键盘上的按键被按下的时候执行哪个硬件中断?
10. 中断处理例程结束时是如何恢复到引发中断状态之前的地方继续执行的?
11. MS-DOS 的哪些功能调用获取和设置中断向量?
12. 解释一下中断处理例程和内存驻留程序之间的不同。
13. 描述一下什么是 TSR 程序。
14. 如何从内存中移除 TSR 程序?
15. 如果内存驻留程序替换了某个中断向量, 那么它该如何利用该中断向量原来的那些处理例程的某些功能?
16. MS-DOS 的哪个功能可以结束一个程序并把部分程序驻留在内存中?
17. 在 No_reset 程序中实际上用什么组合键来重启计算机?

16.5 使用 I/O 端口控制硬件

IA-32 系统提供了两种硬件输入输出方式: 内存映射方式和基于端口的方式。在使用内存映射方式时, 程序直接向特定的内存地址写入数据, 数据则自动传送至设备。类似地, 通过从预定义的内存地址复制数据, 则可以从输入设备读取数据。文本视频显示就是一个内存映射设备的例子, 当在视频内存段写入字符的时候, 字符就会立刻显示在屏幕上。

基于端口的输入输出要求使用 IN 和 OUT 指令从称为“端口”的特定位置读取或写入数据。端口是 CPU 和键盘、扬声器、调制解调器以及声卡等其他设备之间的连接或通道。

16.5.1 输入输出端口

每个输入输出端口都由一个 0~FFFFh 之间的特定数字编号标识。例如,控制扬声器的时候要用一个输入输出端口快速改变扬声器纸盆的状态,也可以通过设置串行端口参数(波特率、奇偶校验等)并通过向串口发送数据来实现与异步适配器的通信。

键盘端口是输入输出端口的典型例子。键被按下的时候,键盘控制器芯片向 60h 端口发送 8 位的扫描码,同时引发硬件中断,这使得 CPU 调用 ROM BIOS 的 INT 9 中断处理例程,INT 9 中断处理例程从端口读入扫描码,查找对应的 ASCII 码,然后把扫描码和 ASCII 码存储在键盘输入缓冲区中。事实上,完全可以绕过操作系统直接从 60h 端口读取字符。

大部分硬件设备除了有数据传输端口之外,还有一些用于监测设备状态及控制设备行为的端口。

IN 和 OUT 指令: IN 指令从端口输入一个字节、字或双字,OUT 指令则相反,它向端口输出一个值,这两条指令的格式如下:

```
IN  累加器, 端口
OUT 端口, 累加器
```

其中端口可以是 0~FFh 之间的一个常量,或者是包含 0~FFFFh 之间的值的 DX 寄存器,累加器必须是 AL (8 位传输),AX (16 位传输)或 EAX (32 位传输)。如下例所示:

```
in  al,3Ch          ; 从端口 3Ch 输入一个字节
out 3ch,al          ; 向端口 3Ch 输出一个字节
mov dx, portNumber  ; DX 可以包含一个端口号
in  ax,dx           ; 从 DX 中包含的端口输入一个字
out dx,ax           ; 向同一端口输出一个字
in  eax,dx          ; 从该端口输入一个双字
out dx,eax          ; 向该端口输出一个双字
```

16.5.2 PC 声音程序

我们可以写一个程序,使用 IN 和 OUT 指令控制 PC 内置的扬声器发出声音。扬声器控制端口 (61h) 通过控制 8255 可编程外围接口芯片开关扬声器:从 61h 端口读入其当前值,把最低两位 (0 位和 1 位) 置位,再把该字节输出到 61h 端口就可以打开扬声器。要关闭扬声器,类似地,首先读入 61h 端口的当前值,再把最低两位清零并输出即可。

在便携式电脑上,如果扬声器直接与声卡相连,而非连接到端口 61h 上,这个声音程序将不能发声。

Intel 8253 定时器控制芯片控制产生声音的频率,方法是向 42h 端口发送一个 0~255 之间的值。下面的扬声器演示程序说明了如何演奏一系列升调音符:

```
TITLE Speaker Demo Program          (Spkr.asm)
; This program plays a series of ascending notes on
; an IBM-PC or compatible computer.

INCLUDE Irvine16.inc

speaker EQU 61h                    ; 扬声器端口的地址
timer   EQU 42h                    ; 定时器端口的地址
delay1  EQU 500
```

```

delay2 EQU 0D000h                ; 音符之间的延时

.code
main PROC
    in     al,speaker              ; 获取扬声器的状态
    push  ax                      ; 保存状态
    or     al,00000011b           ; 低二位位置
    out    speaker,al             ; 打开扬声器
    mov    al,60                  ; 起始频率
L2: out    timer,al               ; 输出脉冲频率到定时器端口
    ; 在频率改变之前创建一个循环
    mov    cx,delay1
L3: push  cx                      ; 外循环
    mov    cx,delay2
L3a:                                ; 内循环
    loop   L3a
    pop    cx
    loop   L3
    sub    al,1                   ; 提高频率
    jnz    L2                     ; 演奏下一个音符
    pop    ax                     ; 获取初始状态
    and    al,11111100b           ; 清除最低二位
    out    speaker,al             ; 关闭扬声器
    exit
main ENDP
END main

```

首先, 程序使用 61h 端口把扬声器状态字节的最低两位置位以打开扬声器:

```

or     al,00000011b              ; 设置最低的两位
out    speaker,al                ; 开启扬声器

```

接着, 程序向计时器芯片发送 60 以设置发音频率:

```

mov     al,60                    ; 开始发音
L2: out    timer,al              ; 计时器端口: 加上扬声器

```

在再次改变频率之前用循环来延时。由于处理器速度不同, 在不同计算机之间延时可能不同, 读者实验时有可能需要调整 delay1 和 delay2 的值:

```

mov     cx,delay1
L3: push  cx                      ; 外循环
    mov    cx,delay2
L3a:                                ; 内循环
    loop   L3a
    pop    cx
    loop   L3

```

延时结束后, 程序把频率控制参数 (1/频率, 频率的倒数) 减 1 以提高发声频率。当循环再次重复的时候, 新的频率控制参数重新发送到定时器。此过程一直重复到 AL 中的频率控制参数等于 0 为止。最后, 程序弹出原来保存的 61h 端口的值, 把最低两位清零关闭扬声器:

```

pop     ax                      ; 获取原始状态
and     al,11111100b           ; 清除最低两位
out     speaker,al             ; 关闭扬声器

```

16.6 本章小结

有时程序员需要显式地定义段, 特别是在要适应使用私有段名的已存代码库的时候。

SEGMENT 和 ENDS 伪指令定义段的开始和结束。当一个段和其他段结合的时候, 对齐方式通知链接器在其后跳过多少空字节, 组合方式通知寄存器如何组合有相同段名的段, 段的类名称则提供了组合段的另一种方式。多个段可以通过使用相同的名字并指定 PUBLIC 组合方式来组合在一起。

ASSUME 伪指令使在编译时计算变量和标号的偏移成为可能。指示处理器在计算有效地址时使用不同于 ASSUME 伪指令指定的另外一个不同的段寄存器。

MS-DOS 命令行处理器解释命令行输入的每条命令。以 COM 和 EXE 为扩展名的程序称为暂留程序。暂留程序被装入内存执行, 然后其占用的内存被释放。在执行时, MS-DOS 在暂留程序前创建一个 256 字节的信息块, 称为程序段前缀。

根据使用扩展名的不同, 暂留程序分为两类: COM 程序和 EXE 程序。COM 程序是未经修改的机器语言程序的映像。存储在磁盘上的 EXE 程序由 EXE 文件头和可装载模块两部分构成。MS-DOS 使用 EXE 程序头中的数据来正确计算段和其他组件的地址。

中断处理例程 (中断服务例程) 简化了输入输出和一些基本的系统任务。读者可以使用自己的代码替换默认的中断处理例程, 以提供更加完整或专业化的服务。中断向量表位于 RAM 的前 1024 字节 (位于 0:0 到 0:03FF 之间)。中断向量表的每个项都由一个 32 位的段-偏移格式的地址构成, 该地址指向中断服务例程。

硬件中断由 8259 可编程控制芯片 (PIC) 产生。PIC 通知 CPU 挂起当前正在执行的程序并执行中断服务例程。硬件中断允许重要的背景事件在丢失重要数据之前引起 CPU 的注意。中断可由多种设备触发, 每种设备都有一个基于中断请求优先级 (IRQ) 的优先级别。

中断允许标志控制着 CPU 是否响应外部 (硬件) 中断。如果该标志置位, 那么中断被允许。如果该标志清零, 中断就被禁止了。使用 STI 和 CLI 指令可以启用 (允许) 和禁止中断。

内存驻留程序 (TSR) 把自己的部分代码驻留在内存中。TSR 程序最常见的用途是用来安装那些要等到机器重启或用特殊的反安装工具移除之前一直保留在内存中的中断服务例程。

No_reset 程序是一个阻止按下 Ctrl-Alt-Delete 键重启系统的 TSR 程序。

IA-32 系统提供了两种硬件输入输出方式: 内存映射方式和基于端口的方式。在使用内存映射方式时, 程序直接向特定的内存地址写入数据, 数据则自动传送至设备。基于端口的输入输出要求使用 IN 和 OUT 指令从称为“端口”的特定位置读取或写入数据。

扬声器控制端口 (61h) 通过控制 8255 可编程外围接口芯片开关扬声器。扬声器演示程序展示了如何演奏一系列升调音符。

第 17 章 浮点处理和指令编码

本章要点

- 浮点二进制表示
- 浮点单元
- Intel 指令编码

17.1 浮点二进制表示

浮点数由三部分构成：符号、尾数和指数。以数字 -1.23154×10^5 为例，其中“-”是符号，表示该浮点数是负数，1.23154 是尾数，5 是指数。

查阅 Intel IA-32 文档：在学习本章的时候要想取得最佳的学习效果，请同时阅读 Intel 的 *IA-32 Intel Architecture Software Developer's Manual*，卷 I 和卷 II 中的相关内容。通过在 Web 浏览器中输入 www.intel.com，然后搜索“IA-32 manuals”，可找到这份文档的免费电子版本。

17.1.1 IEEE 二进制浮点数的表示

Intel 微处理器使用 IEEE 发布的 *Standard 754-1985 for Binary Floating-Point Arithmetic* 中规定的三种浮点二进制存储格式，见表 17.1^①。

表 17.1 IEEE 浮点二进制格式

格 式	说 明
单精度	32 位：符号占 1 位、指数占 8 位、尾数中的小数部分占 23 位。可表示的大致范围是 $2^{-126} \sim 2^{127}$ ，也称为短实数
双精度	64 位：符号占 1 位、指数占 11 位、尾数中的小数部分占 52 位。可表示的大致范围是 $2^{-1022} \sim 2^{1023}$ ，也称为长实数
扩展精度	80 位：符号占 1 位、指数占 16 位、尾数中的小数部分占 63 位。可表示的大致范围是 $2^{-16382} \sim 2^{16383}$ ，也称为扩展实数

三种格式非常相似，因此这里重点讲述单精度格式（参见图 17.1）。图中最高有效位（MSB）在左边：

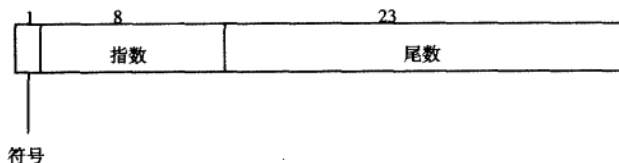


图 17.1 单精度浮点数的格式

符号

二进制浮点数的符号由一个符号位表示，如果该位为 1 表示负数，为 0 表示正数。浮点数 0 是正数。

^① IA-32 *Intel Architecture Software Developer's Manual*，卷 I，第 4 章，同时参见 www.grouper.ieee.org/groups/754/。

尾数

在形如 $m \times b^e$ 的表达式表示的浮点数中, m 称为尾数, b 是基数, e 是指数。尾数部分 m 是一个十进制小数。在第 1 章中讲述二进制、十进制和十六进制数制系统的时候介绍了加权进位表示法的概念, 现在可以把同样的概念扩展到数字的小数部分。例如十进制数 123.154 可表示如下表达式的和:

$$123.154 = (1 \times 10^2) + (2 \times 10^1) + (3 \times 10^0) + (1 \times 10^{-1}) + (5 \times 10^{-2}) + (4 \times 10^{-3})$$

二进制浮点数的表示方法是类似的, 只要用 2 为基数计算位权值即可。例如浮点二进制值 11.1011 可表示为:

$$11.1011 = (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4})$$

另一种表示小数点后面的值的方法是用以 2 的幂为分母的分数之和表示, 下例中和是 11/16 (或 0.6875):

$$.1011 = 1/2 + 0/4 + 1/8 + 1/16 = 11/16$$

得到分数表示的方法是相当直观的。二进制数 1011 实际上是十进制数 11, 可用 1011 (十进制数 11) 做分数的分子。假设 e 是二进制小数点后的有效数字的位数, 那么分母就是 2^e 。在该例中, $e=4$, 因此分母 $2^e=16$ 。表 17.2 给出了另外一些二进制浮点表示转换成十进制分数的例子。表中的最后一项是 23 位规格化尾数所能存储的最小的分数。表 17.3 列出了另外一些例子, 例子中分别给出了二进制浮点数及与其等价的分数和十进制数。

表 17.2 例子: 二进制浮点数转换成分数

二进制浮点数	十进制分数值
11.11	3 3/4
101.0011	5 3/16
1101.100101	13 37/64
0.00101	5/32
1.011	1 3/8
0.000000000000000000000001	1/8388608

表 17.3 二进制和十进制分数

二进制浮点数	十进制分数值	十进制值
.1	1/2	.5
.01	1/4	.25
.001	1/8	.125
.0001	1/16	.062 5
.00001	1/32	.0312 5

尾数的精度

连续实数的整体不可能以有限个数据位的浮点格式表示。例如, 假设使用一种简化的格式表示浮点数, 其中尾数占 5 个数据位, 那么就没有办法表示二进制数 1.1111 和二进制数 10.0000 之间的实数了。因此表示二进制数 1.1111 就需要更高精度的尾数。类似的结论可以推广到 IEEE 双精度格式, 53 位的尾数不能表示需要 54 或更多个数据位的二进制数字。

17.1.2 指数

单精度浮点数的指数是以 8 位无符号整数的格式存储的, 实际存储的是指数值与 127 相加的和。以数字 1.101×2^5 为例, 指数 (5) 与 127 相加的和 (十进制 132, 二进制 10100010) 存储在指数部分。表 17.4 给出了一些指数的例子, 其中的第一列是有符号十进制整数, 第二列是调整后的值, 最后一列是调整后的整数对应的二进制值。调整后的指数部分总是正数 (在 1~254 之间), 实际指数范围是 -126~+127。之所以选择这个范围, 是为了避免最小可能的指数的倒数发生溢出 (这是由于如果最小的指数选择了 -127, 那么 $-127 + 127 = 0$)。

指数 (E)	调整后 (E+127)	二进制值
+5	132	10000100
0	127	01111111
-10	117	01110101
+127	254	11111110
-126	1	00000001
-1	126	01111110

17.1.3 二进制浮点数的正规化

大多数浮点二进制数都是以正规化的格式存储的, 以便能够使得尾数的精度最大化。对于任何给定的浮点二进制数, 可通过移动小数点, 使小数点前仅有一个数字 “1” 从而使其正规化, 指数表示小数点向左 (正数) 或向右移动 (负数) 的位数。下面是一些例子。

未正规化格式	正规化格式
1110.1	1.1101×2^3
.000101	1.01×2^{-4}
1010001.	1.010001×2^6

非正规化值: 二进制浮点数正规化操作的逆过程称为逆正规化, 可通过移动二进制小数点直到指数为零。如果指数部分是正数 n , 那么向右移动小数点 n 位; 如果指数部分是负数 n , 那么向左移动小数点 n 位。如果需要, 开头的空位应以 0 填充。

17.1.4 IEEE 表示法

实数编码

一旦符号位、指数和尾数域进行了正规化和编码, 得到一个完整的二进制 IEEE 短实数就非常容易了。参照图 17.1, 可以把符号位放在最前边, 接下来是指数, 最后是尾数中的小数部分。例如, 二进制值 1.101×2^0 可表示如下:

- 符号位: 0
- 指数: 01111111
- 小数: 1010000000000000000000

调整后的指数 01111111 是十进制数 127, 所有正规化后的二进制尾数在小数点前面都有一个 1, 因此对该位就无需显式编码了。更多的例子如表 17.5 所示。

表 17.5 单精度浮点数值编码的例子

二进制值	调整后的指数值	符号、指数和尾数
-1.11	127	1 01111111 1100000000000000000000
+1101.101	130	0 1000010 1011010000000000000000
-.00101	124	1 01111100 0100000000000000000000
+100111.0	132	0 10000100 0011100000000000000000
+.0000001101011	120	0 01111000 1010110000000000000000

IEEE 规范中还规定了几类实数和非实数的编码：

- 正数 0 和负数 0
- 反向正规化的有限数
- 正规化的有限数
- 正无穷和负无穷
- 非数字值（NaN）
- 不确定数

Intel 浮点单元使用不确定数响应一些无效的浮点操作。

正规化和反向正规化：正规化的有限数是 0 到无穷大之间的所有可以编码为正规化实数的非 0 有限值。尽管乍看起来所有的非零浮点数都可以正规化，不过事实上在值非常接近于 0 时这是不可能的。由于指数范围的限制，FPU 有可能不能把二进制的小数点移动到正规化的位置。假设 FPU 的计算结果是 $1.0101111 \times 2^{-129}$ ，其指数太小无法在单精度实数中存储，这时将产生一个下溢异常，可通过向左移动小数点（每次一位）使其正规化，直到指数落在有效的范围内为止：

$1.010111100000000000001111 \times 2^{-129}$
 $0.101011110000000000000111 \times 2^{-128}$
 $0.010101111000000000000011 \times 2^{-127}$
 $0.001010111100000000000001 \times 2^{-126}$

在这个例子中，由于移动了小数点，精度会有一些损失。

正无穷和负无穷：正无穷（+∞）表示最大的正实数，负无穷（-∞）表示最小的负实数。可以比较无穷值和其他值：-∞ 小于 +∞，-∞ 小于任何有限数，+∞ 大于任何有限数。正无穷和负无穷都能用来表示溢出。

非数字（NaN）：NaN 是不表示任何有效实数的位序列。IA-32 体系结构包含两种类型的 NaN：quiet NaN 可通过大多数算术运算而不会导致任何异常；signalling NaN 可用于产生一个无效操作异常。编译器可以用 Signalling NaN 值填充未初始化的数组，对该数组进行任何计算的企图都会引发一个异常。quiet NaN 可用于存放调试会话产生的诊断信息。浮点单元不会试图对 NaN 执行操作。IA-32 手册详细描述了一套规则，用于确定以这两种类型的 NaN 组合作为操作数时指令执行的结果^①。

特殊值的编码：表 17.6 列出了浮点操作中经常遇到的几个特殊值的编码。标记位的位置既可以是 0，也可以是 1。QNaN 表示 quiet NaN，SNaN 表示 signalling NaN。

表 17.6 特殊的单精度浮点值的编码

值	符号、指数、尾数
正数 0	0 00000000 0000000000000000000000
负数 0	1 00000000 0000000000000000000000

① IA-32 Intel Architecture Software Developer's Manual, 卷 I, 4.8.3 节。

(续表)

值	符号、指数、尾数
正无穷	0 11111111 000000000000000000000000
负无穷	1 11111111 000000000000000000000000
QNaN	x 11111111 1xxxxxxxxxxxxxxxxxxxxxxxxxxx
SNaN	x 11111111 1xxxxxxxxxxxxxxxxxxxxxxxxxxx*

*SNaN的尾数域以0开始,但是其余x表示的域中至少要有个1。

17.1.5 把十进制分数转换为二进制实数

如果十进制分数可以很容易地表示为 $1/2 + 1/4 + 1/8 + \dots$ 的格式,那么就很容易得到其对应的二进制实数。表 17.7 中的左边一列中的大部分分数一眼看去都不容易转换成二进制数,不过如果写成第二列的格式就容易得多了。

表 17.7 十进制分数和二进制实数的例子

十进制分数	分解为	二进制实数
1/2	1/2	.1
1/4	1/4	.01
3/4	1/2 + 1/4	.11
1/8	1/8	.001
7/8	1/2 + 1/4 + 1/8	.111
3/8	1/4 + 1/8	.011
1/16	1/16	.0001
3/16	1/8 + 1/16	.0011
5/16	1/4 + 1/16	.0101

很多实数,如 $1/10$ (0.1) 或 $1/100$ (.01), 不能用有限个二进制数字位表示,这样的分数只能近似表示为若干以 2 的幂为分母的分数之和。想想货币值如 \$39.95 会受什么影响吧!

另一种方法:使用二进制长除法。把十进制分数转换成二进制数时如果涉及到的十进制值比较小,可以使用一种很方便的方法:首先把分子和分母都转换成二进制值,然后再进行除法操作。例如,十进制值 0.5 可表示成分数 $5/10$,十进制 5 是二进制值 0101,十进制值 10 是二进制值 1010,执行下面的除法,我们发现商是二进制值 0.1:

$$\begin{array}{r}
 .1 \\
 1010 \overline{) 01010} \\
 \underline{-1010} \\
 0
 \end{array}$$

在被除数减去 1010 后余数为 0 时,除法终止。因此十进制分数 $5/10$ 等于二进制值 0.1,我们称上面这种方法为二进制长除法^①。

例子:以二进制数表示 0.2。下面把十进制的 0.2 ($2/10$) 转换成二进制值,使用长除法。首先,把二进制值 10 除以 1010 (十进制值 10):

① 来自于 DePaul 大学的 Harvey Nice。

$$\begin{array}{r}
 .00110011 (\text{等等}) \\
 1010 \overline{) 10.00000000} \\
 \underline{1010} \\
 1100 \\
 \underline{1010} \\
 10000 \\
 \underline{1010} \\
 1100 \\
 \underline{1010} \\
 \text{等等}
 \end{array}$$

第一个足够大的余数是 10 000, 在除以 1010 之后, 余数是 110, 在末尾添加一个 0, 新的被除数是 1100, 除以 1010 后, 余数是 10, 再添加 3 个 0, 新的被除数是 10 000, 这等于进行第一次除法操作时被除数的值。从这点开始, 商中重复出现 0011, 因此我们得知准确的商是无法得到的, 0.2 不能用有限个二进制位表示。0.2 的单精度编码的尾数部分是 00110011001100110011001。

把单精度值转换成十进制数

下面是在把 IEEE 单精度值转换成十进制数时的推荐步骤:

1. 如果 MSB (最高有效位) 是 1, 该值为负数, 否则为正数。
2. 接下来的 8 位表示指数, 减去二进制值 01111111 (十进制值 127), 得到未调整的值数值, 把未调整的指数值转换成十进制数。
3. 接下来的 23 位表示尾数, 注意尾数小数点前面有一个被省略的 “1.”, 尾数尾部的 0 可以忽略。这时就可以使用第 1 步中得到的符号位、第二步中得到的指数以及本步骤中得到的尾数创建一个浮点二进制数了。
4. 对第三步得到的浮点数进行逆规格化操作 (根据指数相应地移动小数点, 指数为正则右移, 指数为负则左移)。
5. 从左到右, 使用位权表示方法得到浮点数的 2 的次幂之和的表示, 进而得到相应的十进制数。

例子: 把 IEEE 浮点数 (0 10000010 010110000000000000000000) 转换成十进制数

1. 该数是正数。
2. 未调整的指数值是 00000011, 也就是十进制数 3。
3. 组合符号位、指数和尾数得到二进制数 $+1.01011 \times 2^3$ 。
4. 逆正规化后的二进制数是 $+1010.11$ 。
5. 对应的十进制值是 $+10 \frac{3}{4}$ 或 $+10.75$ 。

17.1.6 本节习题

1. 为什么单精度实数格式不允许以 -127 作为指数?
2. 为什么单精度实数格式不允许以 +128 作为指数?
3. 在 IEEE 双精度格式中, 为尾数的小数部分保留了多少个数据位?
4. 在 IEEE 单精度格式中, 为指数部分保留了多少个数据位?
5. 把二进制浮点数 1101.01101 表示为一个十进制分数。
6. 解释说明为什么十进制数 0.2 不能用有限个数据位表示。
7. 正规化二进制值 11011.01011。

8. 正规化二进制值 0000100111101.1。
9. 给出二进制数+1110.011 的单精度格式的编码。
10. NaN 有哪两种类型?
11. 把分数 5/8 转换成二进制实数。
12. 把分数 17/32 转换成二进制实数。
13. 把十进制值+10.75 转换成 IEEE 单精度实数。
14. 把十进制值-76.0625 转换成 IEEE 单精度实数。

17.2 浮点单元

Intel 8086 处理器是为处理整数运算而设计的, 不过, 事实证明这对于大量使用浮点运算的图形处理程序和运算密集型的程序是个严重的限制。尽管可以采用纯软件仿真的方法模拟浮点运算, 但运算性能会严重下降, 例如像 AutoCAD (Autodesk 公司出品) 这样的软件就需要更加强大的浮点运算支持。Intel 曾经销售过一种名为 8087 的浮点协处理器芯片, 并随处理器的升级换代而升级。从 Intel 486 处理器开始, 浮点处理硬件集成进了主 CPU, 称为浮点单元 (FPU, Floating Point Unit)。

17.2.1 浮点寄存器栈

FPU 不使用通用寄存器 (EAX, EBX 等), FPU 有自己的一套寄存器, 称为寄存器栈。FPU 从内存中把值加载到寄存器栈, 执行计算, 然后再把栈上的值存储到内存中。FPU 指令以后缀格式计算数学表达式, 这和 Hewlett-Packard 计算器差不多。例如, 中缀格式的表达式: $(5 * 6) + 4$ 的后缀格式是:

5 6 * 4 +

中缀表达式 $(A + B) * C$ 使用圆括号覆盖了默认的优先级规则 (默认是先乘法后加法), 但其等价的后缀格式不需要圆括号:

A B + C *

表达式栈: 在对后缀表达式求值的时候可使用堆栈存放中间值。图 17.2 显示了对表达式 “5 6 * 4 +” 求值所需的步骤, 堆栈项以 ST(0) 和 ST(1) 标识, ST(0) 通常表示堆栈指针指向的位置 (栈顶)。

把中缀表达式转换成后缀表达式的常用方法在计算机科学的入门教科书以及在因特网上都有很多很好的介绍, 这里就不再重复了。表 17.8 给出了一些中缀表达式和后缀表达式的例子。

表 17.8 中缀表达式转换成后缀表达式的例子

中缀格式	后缀格式
A + B	A B +
(A - B) / D	A B - D /
(A + B) * C + D)	A B + C D + *
((A + B) / C) * (E - F)	A B + C / E F - *

从左到右	堆栈	动作
5	<div>5</div> ST(0)	push 5
5 6	<div>5</div> ST(1) <div>6</div> ST(0)	push 6
5 6 *	<div>30</div> ST(0)	Multiply ST(1) by ST(0) and pop ST(0) off the stack.
5 6 * 4	<div>30</div> ST(1) <div>4</div> ST(0)	push 4
5 6 * 4 -	<div>26</div> ST(0)	Subtract ST(0) from ST(1) and pop ST(1) off the stack

图 17.2 对后缀表达式“5 6 * 4 -”求值

浮点数据寄存器

FPU 有 8 个可独立寻址的 80 位寄存器，分别名为 R0, R1, ..., R7（如图 17.3 所示），它们以堆栈形式组织在一起，统称为寄存器栈。栈顶由 FPU 状态字中的一个名为 TOP 的域（占三个二进制位）来标识，对寄存器的引用都是相对于栈顶而言的。例如在图 17.3 中，TOP 等于二进制值 011，说明 R3 是栈顶。在编写浮点指令时栈顶也写为 ST(0)（或 ST），最后一个寄存器（栈底）写为 ST(7)。

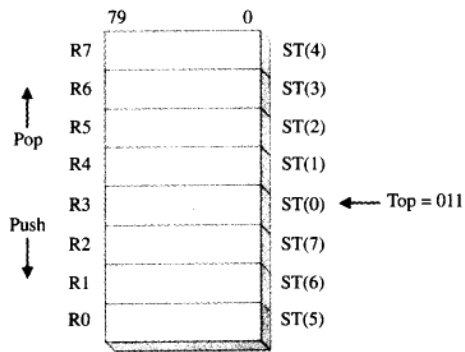


图 17.3 浮点数据寄存器堆栈

跟预期的一致，压栈（push）操作（也称为加载操作）把指示栈顶的 TOP 域值减 1 并把结果复制到由 ST(0)标识的寄存器中，如果在压栈操作之前 TOP 等于 0，压栈操作使 TOP 回滚指示寄存器 R7。出栈（pop）操作（也称为存储操作）把 ST(0)中的数据复制到一个操作数中并把 TOP 域的值增 1。如果在出栈之前 TOP 等于 7，出栈操作使 TOP 回滚指示寄存器 R0。加载一个值至浮点栈中时如果会覆盖已存在的数据，就会产生一个浮点异常。图 17.4 显示了同一浮点栈在 1.0 和 2.0 压栈后的情况。

浮点寄存器中的值以 10 字节的 IEEE 扩展实数格式（也称为临时实数格式）存储，FPU 在内存中保存算术运算的结果时，自动把结果转换为以下格式之一：整数、长整数、单精度（短实数）、双精度（长实数）或压缩的二进制编码的十进制整数。

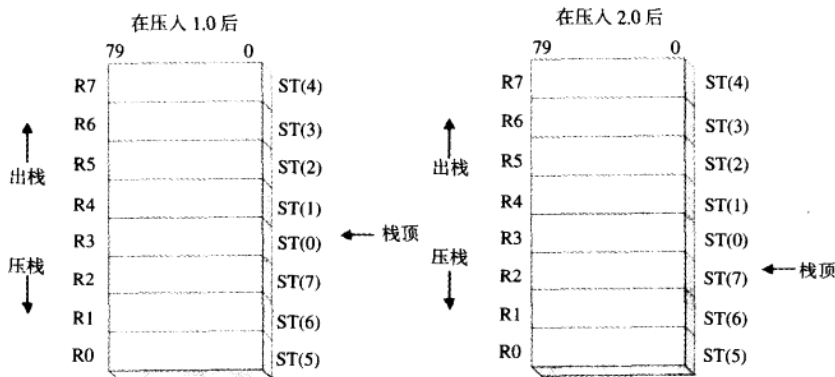


图 17.4 在压入 1.0 和 2.0 后的浮点栈

特殊用途寄存器

FPU 有 6 个特殊用途的寄存器（如图 17.5 所示）。

- 一个 10 位的操作码寄存器：存储最后一条执行的非控制指令。
- 一个 16 位的控制寄存器：在执行计算时控制 FPU 的精度和使用的近似方法。
- 一个 16 位的状态寄存器：存放着栈顶指针、条件码以及关于异常的警告。
- 一个 16 位的标记字寄存器：指示 FPU 数据寄存器栈中每个寄存器的内容的状态，每个寄存器使用两位指示寄存器是否包含一个有效的数字、零、特殊值（NaN、无穷数、反规格化或不支持的格式）或是否为空。
- 一个 48 位的最后指令指针寄存器：存放最后执行的非控制指令的指针。
- 一个 48 位的最后数据（操作数）指针寄存器：存放最后执行的指令使用的数据操作数（如果有的话）。

操作系统在任务切换时使用这些特殊用途的寄存器保存浮点单元的状态信息。我们在第 2 章讲述 CPU 如何执行多任务的时候提到过任务状态的保存。

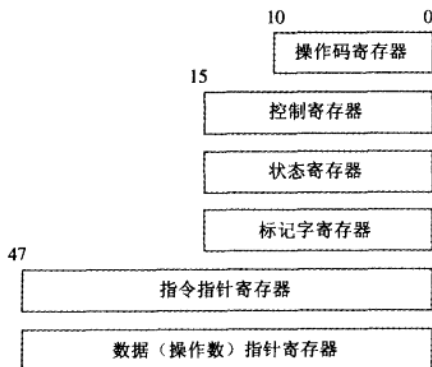


图 17.5 FPU 的特殊用途寄存器

17.2.2 近似

FPU 在进行浮点计算时试图产生准确的结果，不过在许多情况下这是不可能的，因为目的操作数根本就不能准确表示计算的结果。例如，假设某种存储格式只允许 3 个小数位，这种格式就

只能存储 1.011 或 1.101, 但不能存储 1.0101。如果计算产生的精确结果是 +1.0111 (十进制数 1.4375), 我们就必须通过加 .0001 或减去 .0001 向上或向下近似:

- (a) 1.0111 --> 1.100
(b) 1.0111 --> 1.011

如果精确结果是负数, 那么加 -.0001 会使近似值趋向 $-\infty$, 减去 -.0001 会使近似值趋向 0 和 $+\infty$ 。

FPU 允许选择下面 4 种近似方法。

- 近似到最接近的偶数: 近似结果最接近准确结果, 如果有两个值与精确结果近似度相同, 则选取最接近的偶数 (最低有效位是 0)。
- 向下近似趋向于 $-\infty$: 近似结果小于或等于精确结果。
- 向上近似趋向于 $+\infty$: 近似结果大于或等于精确结果。
- 近似趋向于 0: 也称为剪裁, 近似结果的绝对值小于或等于精确结果。

FPU 控制字: FPU 的状态字中包含了一个名为 RC 的域, 包含两个数据位, 该域指定使用何种近似方法。该域取值如下:

- 二进制值 00: 近似到最近的偶数 (默认)。
- 二进制值 01: 向下近似趋向于 $-\infty$ 。
- 二进制值 10: 向上近似趋向于 $+\infty$ 。
- 二进制值 11: 近似趋向于 0 (剪裁)。

近似到最接近的偶数是默认的, 被认为是最准确的, 适合大多数应用程序。表 17.9 给出了二进制值 +1.0111 如何应用 4 种近似方法的例子。类似地, 表 17.10 给出了二进制值 -1.0111 可能的近似值。

表 17.9 例子: +1.0111 的近似

方 法	精确结果	近似后
近似到最近的偶数	1.0111	1.100
向下近似趋向于 $-\infty$	1.0111	1.011
向上近似趋向于 $+\infty$	1.0111	1.100
近似趋向于 0 (剪裁)	1.0111	1.011

表 17.10 例子: -1.0111 的近似

方 法	精确结果	近似后
近似到最近的偶数	-1.0111	-1.100
向下近似趋向于 $-\infty$	-1.0111	-1.100
向上近似趋向于 $-\infty$	-1.0111	-1.011
近似趋向于 0 (剪裁)	-1.0111	-1.011

17.2.3 浮点异常

每个程序都有可能出错, FPU 必须能够处理错误。FPU 能够识别和检查 6 种类型的异常: 无效操作 (#I)、除零 (#Z)、反规格化操作数 (#D)、数值溢出 (#O)、数值下溢 (#U)、无效精度 (#P)。前三种异常 (#I, #Z, #D) 是在算术操作发生之前检查的, 后三种异常 (#O, #U, #P) 是在算术操作发生之后检查的。

每种异常类型都有相应的标志位和掩码, 检测到浮点异常时, 处理器设置相应的标志位。对于标识出来的每种异常, 根据其掩码位的不同, 处理器可采取两种不同的动作:

- 如果相应的掩码位置位，处理器自动处理异常并允许程序继续。
- 如果相应的掩码位清零，处理器调用软件异常处理程序。

处理器的掩码响应方式对大多数程序而言都是可接受的。定制的异常处理程序可在应用程序要响应特定异常的情况下使用。单条指令可引发多个异常，因此处理器保留自上次异常清除后发生的异常记录，因此在一系列计算完成之后，可以检查是否发生了异常。

17.2.4 浮点指令集

浮点指令集有点复杂，因此本节尽量给读者一个关于其功能的大致印象，可能会给出一些例子用以说明编译器通常会生成的代码是个什么样子。除此之外，本节还会讲述如何修改近似模式控制 FPU。浮点指令集包含下面几类指令：

- 数据传送指令
- 基本的算术运算指令
- 比较指令
- 超越指令
- 常量加载指令（特殊的预定义常量）
- x87 FPU 控制指令
- x87 FPU 和 SIMD 状态管理指令

浮点指令总是以字母 F 开头，以便与 CPU 指令区别开。指令的第二个字母（通常是 B 或 I）说明了内存操作数应如何解释：B 表示二/十进制（BCD）操作数，I 表示二进制整数操作数。如果未指定 B 或 I，就表示操作数是实数格式的。例如 FBLD 对 BCD 数进行操作，FILD 对整数进行操作，FLD 对实数进行操作。

附录 B 中包含了 IA-32 浮点指令的参考。

操作数：浮点指令最多可以有两个操作数，也可以无操作数或只有一个操作数。如果有两个操作数，其中的一个必须是浮点寄存器。没有立即数操作数，但可加载一些预定义的值（如 0.0， π 以及 1b 10 等）。通用寄存器 EAX，EBX，ECX，EDX 等不能作为操作数，不允许内存到内存的操作。

整数操作数必须从内存加载至 FPU（绝不能从 CPU 寄存器中加载），FPU 把整数自动转换成浮点数的格式。类似地，在内存中存储浮点值的时候，浮点值自动剪裁或近似取整。

初始化（FINIT）

FINIT 指令初始化浮点单元，把 FPU 的控制字设为 037Fh，掩盖所有的浮点异常，把近似方法设置为最接近的偶数，并把计算精度设为 64 位。强烈建议在程序的开始调用 FINIT，以使 FPU 处于一个固定的初始状态。

浮点数据类型

下面快速回顾一下 MASM 支持的浮点数据类型（QWORD，TBYTE，REAL4，REAL8，REAL10），如表 17.11 所示。在定义 FPU 指令使用的数据类型时，会用到这些数据类型。例如，在加载一个浮点变量至 FPU 堆栈时，变量可定义为 REAL4，REAL8，REAL10：

```
.data
bigVal REAL10 1.212342342234234243E+864
.code
fld bigVal ; 加载变量至浮点栈
```


表 17.11 内部数据类型

类 型	用 途
QWORD	64 位整数
TBYTE	80 位 (10 字节) 整数
REAL4	32 位 (4 字节) IEEE 短实数
REAL8	64 位 (8 字节) IEEE 长实数
REAL10	80 位 (10 字节) IEEE 扩展实数

加载浮点值 (FLD)

FLD (加载浮点值) 指令复制一个浮点数至 FPU 的栈顶[即 ST(0)], 操作数可以是 32 位、64 位或 80 位的内存操作数 (REAL4, REAL8, REAL10) 或另外一个浮点寄存器:

```
FLD m32fp
FLD m64fp
FLD m80fp
FLD ST(i)
```

内存操作数的类型: FLD 支持的内存操作数的类型和 MOV 是一样的。下面是一些例子:

```
.data
array REAL8 10 DUP(?)
.code
fld array           ; direct
fld [array+16]      ; direct-offset
fld REAL8 PTR[esi]  ; indirect
fld array[esi]       ; indexed
fld array[esi*8]     ; indexed, scaled
fld array[esi*TYPE array] ; indexed, scaled
fld REAL8 PTR[ebx+esi] ; base-index
fld array[ebx+esi]   ; base-index-displacement
fld array[ebx+esi*TYPE array] ; base-index-displacement, scaled
```

例子: 下面的例子加载两个直接操作数至 FPU 堆栈:

```
.data
dblOne REAL8 234.56
dblTwo REAL8 10.1
.code
fld dblOne           ; ST(0) = dblOne
fld dblTwo           ; ST(0) = dblTwo, ST(1) = dblOne
```

下图显示了在每条指令执行后堆栈的内容:

fld dblOne	ST(0)	234.56
fld dblTwo	ST(1)	234.56
	ST(0)	10.1

在第二条 FLD 执行的时候, TOP 域减 1, 导致原来标记为 ST(0) 的元素变成了 ST(1)。

FILD: FILD 指令把 16 位、32 位、64 位的整数源操作数转换成双精度浮点数并把其加载到 ST(0), 源操作数的符号位保留, 在 17.2.10 节 (混合模式算术运算) 将详细解释该指令。FILD 支持的内存操作数类型 (间接操作数、变址操作数、基址变址操作数等) 同 MOV 指令。

加载常量：下面的指令在堆栈上加载特定的常量，这些指令无操作数：

- FLDI 指令在寄存器堆栈上压入 1.0。
- FLD L2T 指令在寄存器堆栈上压入 $\lg 10$ 。
- FLD L2E 指令在寄存器堆栈上压入 $\lg e$ 。
- FLDPI 指令在寄存器堆栈上压入 π 。
- FLDLG2 指令在寄存器堆栈上压入 $\lg 2$ 。
- FLDLN2 指令在寄存器堆栈上压入 $\ln 2$ 。
- FLDZ 指令在寄存器堆栈上压入 0.0。

存储浮点值 (FST, FSTP)

FST 指令 (存储浮点值) 复制 FPU 的栈顶的操作数至内存中，操作数可以是 32 位、64 位或 80 位的内存操作数 (REAL4, REAL8, REAL10) 或另外一个浮点寄存器：

```
FST m32fp
FST m64fp
FST ST(i)
```

FST 不会弹出栈顶元素，下面的指令把 ST(0) 存储到内存中，假设 ST(0) 等于 10.1 并且 ST(1) 等于 234.56：

```
fst dblThree           ; 10.1
fst dblFour            ; 10.1
```

单凭直观感觉，我们可能会期望 dblFour 等于 234.56，不过由于第一条 FST 指令把 10.1 留在 ST(0) 中，因此结果正好相反。如果想把 ST(1) 复制到 dblFour 中，第一条指令就必须用 FSTP。

FSTP：FSTP (存储浮点值并出栈) 复制 ST(0) 至内存中并弹出 ST(0)，假设在执行下面的指令之前 ST(0) 等于 10.1 并且 ST(1) 等于 234.56：

```
fstp dblThree          ; 10.1
fstp dblFour           ; 234.56
```

在执行之后，从逻辑上来讲这两个值已经从堆栈上移除了。从物理上来讲，每次 FSTP 指令执行后，TOP 指针增 1，改变了 ST(0) 的位置。

FIST (存储整数) 指令把 ST(0) 中的值转换成有符号整数并把结果存储到目的操作数中，值可以存储在字或双字中。17.2.10 节 (混合模式算术运算) 将讲述其详细用法。FIST 支持的内存操作数格式同 FST。

17.2.5 算术运算指令

基本的算术运算指令如表 17.12 所示，算术运算指令支持的内存操作数类型同 FLD (加载) 和 FST (存储)，因此操作数类型可以是间接操作数、变址操作数、基址变址操作数等。

表 17.12 基本的算术运算指令

指 令	说 明
FCHS	改变符号
FADD	源和目的相加
FSUB	从目的中减去源
FSUBR	源乘以目的
FMUL	目的除以源
FDIV	源除以目的
FDIVR	目的除以源

FCHS 和 FABS

FCHS (改变符号) 指令把 ST(0) 中的值的符号变反, FABS (绝对值) 指令取 ST(0) 中值的绝对值, 这两条指令都不需要操作数:

```
FCHS
FABS
```

FADD, FADDP, FIADD

FADD (加法) 指令的格式如下, 其中 m32fp 是一个 REAL4 类型的内存操作数, m64fp 是一个 REAL8 类型的操作数, i 是寄存器号:

```
FADD 1
FADD m32fp
FADD m64fp
FADD ST(0), ST(i)
FADD ST(i), ST(0)
```

无操作数: 如果 FADD 不带操作数, 那么 ST(0) 和 ST(1) 相加, 结果临时存储在 ST(1) 中, 然后 ST(0) 弹出堆栈, 最终结果存储在栈顶。下图解释说明了无操作数的 FADD 指令执行的情况, 假设堆栈中已经包含了两个变量。

fadd	之前	ST(1)	234.56
		ST(0)	10.1
	之后	ST(0)	244.66

寄存器操作数: 假设浮点栈的内容和上例相同, 下图解释了 ST(0) 和 ST(1) 相加的过程。

fadd st(1), st(0)	之前	ST(1)	234.56
		ST(0)	10.1
	之后	ST(1)	244.66
		ST(0)	10.1

内存操作数: FADD 在使用内存操作数时, 把操作数和 ST(0) 相加, 下面是一些例子:

```
fadd mySingle           ; ST(0) += mySingle
fadd REAL8 PTR[esi]     ; ST(0) += [esi]
```

FADDP: FADDP 指令 (相加并出栈) 指令在执行完加法后从堆栈上弹出 ST(0), 其格式如下:

```
FADDP ST(i), ST(0)
```

下图解释了 FADDP 是如何工作的。

① MASM 使用无参数的 FADD 指令执行和无参数的 Intel FADDP 指令同样的操作。

faddp st(1),st(0)

之前

ST(1)

234.56

ST(0)

10.1

之后

ST(0)

244.66

FIADD: FIADD (与整数相加) 指令把源操作数转换成扩展精度浮点数格式, 然后再和 ST(0) 相加。其格式如下:

```
FIADD m16int
FIADD m32int
```

例子:

```
.data
myInteger DWORD 1
.code
fiadd myInteger          ; ST(0) += myInteger
```

FSUB, FSUBP, FISUB

FSUB 指令从目的操作数中减去源操作数, 把差存储在目的操作数中。目的应总是一个 FPU 寄存器, 源可以是 FPU 寄存器或内存操作数, 其操作数格式同 FADD:

```
FSUB①
FSUB m32fp
FSUB m64fp
FSUB ST(0), ST(i)
FSUB ST(i), ST(0)
```

除了执行的是减法操作而不是加法操作之外, FSUB 的操作和 FADD 很相似。例如, 无操作数的 FSUB 从 ST(1) 中减去 ST(0), 结果临时存储在 ST(1) 中, 然后从堆栈中弹出 ST(0), 这样最后的结果就保存在栈顶了。FSUB 在使用内存操作数时从 ST(0) 中减去内存操作数, 不会弹出栈顶元素:

```
fsub mySingle          ; ST(0) -= mySingle
fsub array[edi*8]      ; ST(0) -= array[edi*8]
```

FSUBP: FSUBP (相减并出栈) 指令在执行完减法后从堆栈中弹出 ST(0)。MASM 支持下面的格式:

```
FSUBP ST(i),ST(0)
```

FISUB: FISUB (减整数) 指令把源操作数转换成扩展精度的浮点数格式, 然后再从 ST(0) 中减去源操作数:

```
FISUB m16int
FISUB m32int
```

FMUL, FMULP, FIMUL

FMUL 指令把源操作数和目的操作数相乘, 结果存储在目的操作数中。目的应总是 FPU 寄存器, 源可以是寄存器或内存操作数。其格式同 FADD 和 FSUB 是一样的:

```
FMUL②
```

① MASM 使用无参数的 FSUB 指令执行和无参数的 Intel FSUBP 指令同样的操作。

② MASM 使用无参数的 FMUL 指令执行和无参数的 Intel FMULP 指令同样的操作。

```
FMUL m32fp
FMUL m64fp
FMUL ST(0), ST(i)
FMUL ST(i), ST(0)
```

除执行的操作是乘法而不是加法之外, FMUL 的操作和 FADD 非常相似。例如, 无操作数的 FMUL 把 ST(1)和 ST(0)相乘, 积临时存储在 ST(1)中, 然后从堆栈中弹出 ST(0), 这样最后的结果就保存在栈顶了。FMUL 在使用内存操作数时把 ST(0)和内存操作数相乘, 不会弹出栈顶元素:

```
fmul mySingle ; ST(0) *= mySingle
```

FMULP: FMULP (相乘并出栈) 指令在执行完乘法后从堆栈中弹出 ST(0)。MASM 支持下面的格式:

```
FMULP ST(i), ST(0)
```

FIMUL 与 FIADD 的格式基本相同, 不过执行的操作是乘法而非加法:

```
FIMUL m16int
FIMUL m32int
```

FDIV, FDIVP, FIDIV

FDIV 指令把源操作数和目的操作数相除, 结果存储在目的操作数中。目的应总是 FPU 寄存器, 源可以是寄存器或内存操作数。其格式同 FADD 和 FSUB 是一样的:

```
FDIV①
FDIV m32fp
FDIV m64fp
FDIV ST(0), ST(i)
FDIV ST(i), ST(0)
```

除执行的操作是除法而不是加法之外, FDIV 的操作和 FADD 非常相似。例如, 无操作数的 FDIV 把 ST(1)和 ST(0)相除, 商临时存储在 ST(1)中, 然后从堆栈中弹出 ST(0), 这样最后的结果就保存在栈顶了。FDIV 在使用内存操作数时把 ST(0)和内存操作数相除, 不会弹出栈顶元素。下面的代码中 dblOne 除以 dblTwo, 商存储在 dblQuot 中:

```
.data
dblOne REAL8 1234.56
dblTwo REAL8 10.0
dblQuot REAL8 ?
.code
fld dblOne ; 加载至 ST(0)
fdv dblTwo ; ST(0)除以 dblTwo
fstp dblQuot ; 把 ST(0)存储到 dblQuot 中
```

如果源操作数是 0, 就会产生一个除零异常。有很多特殊情况, 如正无穷、负无穷、正数 0、负数 0、NaN 作为被除数时, 其细节请参阅 IA-32 指令集参考手册。

FIDIV: FIDIV 指令把整数源操作数转换成扩展精度的浮点数据格式, 然后再把 ST(0)和源操作数相除, 格式如下:

```
FIDIV m16int
FIDIV m32int
```

17.2.6 浮点值的比较

浮点值的比较不能使用 CMP 指令 (执行比较时使用整数减法操作), 应该使用 FCOM 指令。

① MASM 使用无参数的 FDIV 指令执行和无参数的 Intel FDIVP 指令同样的操作。

在执行完 FCOM 指令之后、使用条件跳转指令 (JA, JB, JE 等) 之前还要执行一些必需的指令。

FCOM, FCOMP, FCOMPP: FCOM 指令 (比较浮点值) 比较 ST(0) 和源操作数, 源操作数可以是内存操作数或 FPU 寄存器, 其格式如下所示。

指 令	描 述
FCOM	比较 ST(0) 和 ST(1)
FCOM <i>m32fp</i>	比较 ST(0) 和 <i>m32fp</i>
FCOM <i>m64fp</i>	比较 ST(0) 和 <i>m64fp</i>
FCOM ST(<i>i</i>)	比较 ST(0) 和 ST(<i>i</i>)

FCOMP 的操作数格式和 FCOM 相同, 对于每种类型的操作数, FCOMP 执行的动作和 FCOM 基本相同, 不过最后还要从堆栈上弹出 ST(0)。FCOMPP 和 FCOMP 基本相同, 最后还要再一次从堆栈上弹出 ST(0)。

条件码: C3, C2, C0 这三个 FPU 条件码标志说明了浮点值比较的结果 (如表 17.13 所示), 表格的标题栏中列出了各个浮点标志对应的 CPU 状态标志, 这是因为 C3, C2, C0 分别与零标志、奇偶标志和进位标志在功能上类似。

表 17.13 FCOM, FCOMP, FCOMPP 设置的条件码

条 件	C3 (零标志)	C2 (奇偶标志)	C0 (进位标志)	应使用的条件跳转指令
ST(0) > SRC	0	0	0	JA, JNBE
ST(0) < SRC	0	0	1	HB, JNAE
ST(0) = SRC	1	0	0	JE, JZ
无序*	1	1	1	(无)

*如果抛出了无效算术操作数异常 (由于无效操作数) 并且异常被屏蔽掉了, 那么 C3, C2, C0 的值根据该行设置。

在比较了两个值并设置了 FPU 条件码之后, 主要的挑战在于找到一种方法以根据条件码分支跳转到目标号处, 这涉及到两个步骤:

- 使用 FNSTSW 指令把 FPU 状态字送 AX。
- 使用 SAHF 指令把 AH 复制到 EFLAGS 寄存器中。

一旦条件码复制到 EFLAGS 寄存器之后, 就可以使用基于零标志、奇偶标志和进位标志的跳转指令了。表 17.13 列出了对于每组条件码的组合可使用的合适的条件跳转指令。对于其他的条件码的组合, 还可以使用其他的条件跳转指令, 如 JAE 指令在 CF = 0 时跳转; JBE 在 CF = 1 或 ZF = 1 时跳转; JNE 在 ZF = 0 时跳转。

例子: 假设有下面的 C++ 代码:

```
double X = 1.2;
double Y = 3.0;
int N = 0;
if( X < Y )
    N = 1;
```

下面是等价的汇编语言代码:

```
.data
X REAL8 1.2
Y REAL8 3.0
N DWORD 0
.code
```

```

; if( X < Y )
;   N = 1
    fld     X                ; ST(0) = X
    fcomp   Y                ; compare ST(0) to Y
    fnstsw  ax               ; move status word into AX
    sahf    ; copy AH into EFLAGS
    jnb     L1               ; X not < Y? skip
    mov     N,1              ; N = 1
L1:

```

P6 的改进：对于前面的例子，有一点值得注意：浮点数比较比整数比较运行时开销更大，因此 Intel 的 P6 系列处理器引入了 FCOMI 指令，该指令比较两个浮点值并直接设置零标志、奇偶标志和进位标志（P6 系列处理器始于奔腾 Pro 和奔腾 II 处理器）。FCOMI 的格式如下：

FCOMI ST(0),ST(i)

下面使用 FCOMI 指令重写前面的例子代码（比较 X 和 Y）：

```

.code
; if( X < Y )
;   N = 1
    fld     Y                ; ST(0) = Y
    fld     X                ; ST(0)= X, ST(1)= Y
    fcomi   ST(0),ST(1)      ; compare ST(0) to ST(1)
    jnb     L1               ; ST(0) not < ST(1)? skip
    mov     N,1              ; N = 1
L1:

```

FCOMI 指令替代了前面例子中的三条指令，不过需要一条额外的 FLD 指令。FCOMI 指令不接受内存操作数。

比较是否相等

几乎所有的程序入门设计教材都会警告读者不要去比较浮点值是否相等，这是由于计算过程中的近似可能导致错误。这个问题可通过计算下面的表达式说明： $(\text{sqrt}(2.0) * \text{sqrt}(2.0)) - 2.0$ 。从数学上讲，这个表达式的结果应该是 0，但实际的结果却不是（大约为 $4.4408921\text{E}-016$ ）。表 17.14 使用下面的数据定义展示了每步之后的 FPU 堆栈：

```
val1 REAL8 2.0
```

表 17.14 计算 $(\text{sqrt}(2.0) * \text{sqrt}(2.0)) - 2.0$

指 令	FPU 堆栈
fld val1	ST(0): +2.0000000E+000
fsqrt	ST(0): +1.4142135E+000
fmul ST(0), ST(0)	ST(0): +2.0000000E+000
fsud val1	ST(0): +4.4408921E+016

比较浮点值 x 和 y 是否相等的正确做法是取其差值的绝对值 $|x - y|$ 并和用户自定义的一个小的正数相比较。下面是实现类似功能的汇编语言代码，其中使用了一个小正数作为在认为这两个值相等时其差值的临界值：

```

.data
epsilon REAL8 1.0E-12
val2 REAL8 0.0                ; value to compare
val3 REAL8 1.001E-13          ; considered equal to val2

.code
; if( val2 == val3 ), display "Values are equal".

```

```

fld    epsilon
fld    val2
fsub   val3
fabs
fcomi  ST(0),ST(1)
ja     skip
mWrite <"Values are equal",0dh,0ah>
skip:

```

表 17.15 跟踪了程序指令的执行过程，显示了每条指令执行后浮点栈的情况。

表 17.15 程序的执行过程

指 令	FPU 堆栈
fld epsilon	ST(0): +1.0000000E-012
fld val2	ST(0): +0.0000000E+000
	ST(1): +1.0000000E-012
fsub val3	ST(0): -1.0010000E-013
	ST(1): +1.0010000E-012
fabs	ST(0): +1.0010000E-013
	ST(1): +1.0000000E-012
fcomi ST(0),ST(1)	ST(0) < ST(1); CF = 1, ZF = 0

如果重新定义 val3，使其大于临界值，则 val3 和 val2 将不再相等：

```
val3 REAL8 1.001E-12 ; not equal
```

17.2.7 读写浮点值

本书附带的链接库中包含了两个处理浮点数输入输出的过程，它们是由 San Jose State University 的 William Barrett 编写的。

- ReadFloat: 从键盘读入一个浮点值并把它压入浮点栈。
- WriteFloat: 在控制台窗口上以指数格式显示 ST(0)的值。

ReadFloat 接收多种格式的浮点数。下面是一些例子：

```

35
+35.
-3.5
.35
3.5E5
3.5E005
-3.5E+5
3.5E-4
+3.5E-4

```

ShowFPUStack: 显示浮点堆栈的内容，这是由 Pacific Lutheran University 的 James Brink 编写的。调用该过程时无需参数：

```
call ShowFPUStack
```

例子程序：下面的例子程序在 FPU 堆栈上压入两个浮点值，然后显示，接下来读入两个用户输入的值，相乘并显示其乘积：

```

TITLE 32-bit Floating-Point I/O Test (floatTest32.asm)
INCLUDE Irvine32.inc
INCLUDE macros.inc

```



```

.data
first REAL8 123.456
second REAL8 10.0
third REAL8 ?

.code
main PROC
    finit                    ; 初始化 FPU
; Push two floats and display the FPU stack.
    fld first
    fld second
    call ShowFPUStack

; 输入两个浮点数并显示其乘积
    mWrite "Please enter a real number: "
    call ReadFloat
    mWrite "Please enter a real number: "
    call ReadFloat
    fmul ST(0),ST(1)         ; 相乘
    mWrite "Their product is: "
    call WriteFloat
    call CrLf

    exit
main ENDP
END main

```

输入输出的示例如下 (用户输入以粗体显示):

```

----- FPU Stack -----
ST(0): +1.0090000E+001
ST(1): +1.2345600E+002
Please enter a real number: 3.5
Please enter a real number: 4.2
Their product is: +1.4700000E+001

```

17.2.8 异常的同步

CPU 和 FPU 分别是独立的单元, 因此浮点指令可以和整数及系统指令同时执行, 这称为并行 (concurrency)。并行执行浮点指令在发生未屏蔽的异常时可能会导致问题, 屏蔽的异常不会导致问题, 因为 FPU 总是会执行完当前的操作并存储结果。

未屏蔽的异常发生时, 当前执行的浮点指令被中断, FPU 产生异常事件信号。下一条浮点指令或 FWAIT (WAIT) 指令要执行时, FPU 检查是否有未决异常, 如果有, 则调用浮点异常处理程序 (一个子过程)。

如果产生异常的浮点指令后跟的是一条整数指令或系统指令又会怎样呢? 遗憾的是, 这些指令不会检查未决异常——它们将立即执行。假设第一条浮点指令要把其输出存储到一个内存操作数中, 第二条整数指令修改该内存操作数, 如果第一条指令发生了异常, 那么异常处理程序就不能适时执行了, 这通常会导致错误的结果。下面是一个例子:

```

.data
intVal DWORD 25
.code
fld intVal                ; 存储 ST(0) 至 intVal
inc intVal                ; 整数值增 1

```

WAIT 和 FWAIT 指令正是用来强制处理器在执行下一条指令之前检查未决的未屏蔽浮点异常的, 这两条指令都能解决这里潜在的同步问题。在下例中, 直到异常处理程序执行完后 INC 指令才能执行:

```
fild intVal          ; 存储 ST(0) 至 intVal
fwait               ; 等待未决异常
inc intVal          ; 整数值增 1
```

17.2.9 代码示例

本节看几个演示浮点算术运算指令的例子。一种好的学习方法是使用 C++ 编写表达式, 然后编译, 再查看编译器生成的代码。

表达式

下面编码实现表达式 $valD = -valA + (valB * valC)$ 。按部就班的方法是: 加载 $valA$ 至浮点栈并求反, 加载 $valB$ 至 $ST(0)$, 这时 $-valA$ 保存在 $ST(1)$ 中, $valC$ 和 $ST(0)$ 相乘, 乘积保存在 $ST(0)$ 中, $ST(0)$ 和 $ST(1)$ 相加并把和存储在 $valD$ 中。实现代码如下:

```
.data
valA REAL8 1.5
valB REAL8 2.5
valC REAL8 3.0
valD REAL8 ?; +6.0
.code
fld valA          ; ST(0)=valA
fchs             ; 改变 ST(0) 中值的符号
fld valB          ; 加载 valB 至 ST(0)
fmul valC        ; ST(0)*=valC
fadd             ; ST(0)+=ST(1)
fstp valD         ; 存储 ST(0) 至 valD 中
```

数组之和

下面的代码计算一个双精度实数数组之和并显示:

```
ARRAY_SIZE = 20
.data
sngArray REAL8 ARRAY_SIZE DUP(?)
.code
mov esi, 0          ; 数组的索引
fldz               ; 在浮点栈上压入 0.0
mov ecx, ARRAY_SIZE
L1: fld sngArray[esi] ; 加载内存操作数至 ST(0)
fadd              ; ST(0) 和 ST(1) 相加 后 ST(0) 出栈
add esi, TYPE REAL8 ; 下一个数组元素
loop L1
call WriteFloat     ; 显示 ST(0) 中的和
```

平方根之和

FSQRT 指令计算 $ST(0)$ 的平方根并把结果存储在 $ST(0)$ 中, 下面的代码计算了两个平方根之和:

```
.data
valA REAL8 25.0
valB REAL8 36.0
.code
fld valA          ; push valA
```

```

fsqrt                ; ST(0) = sqrt(valA)
fld  valB             ; push valB
fsqrt                ; ST(0) = sqrt(valB)
fadd                 ; add ST(0), ST(1)

```

数组的点积

下面的代码计算表达式 $(array[0] * array[1]) + (array[2] * array[3])$, 这种计算有时也称为点积 (dot product)。表 17.16 显示了每条指令执行后 FPU 栈的内容。下面是输入数据:

```

.data
array REAL4 6.0, 2.0, 4.5, 3.2

```

表 17.16 计算点积: $(6.0 * 2.0) + (4.5 * 3.2)$

指 令	FPU 堆栈
fld array	ST(0): +6.000000E+000
fmul [array + 4]	ST(0): +1.200000E+001
fld [array + 8]	ST(0): +4.500000E+000
	ST(1): +1.200000E+001
fmul [array + 12]	ST(0): +1.440000E+001
	ST(1): +1.200000E+001
fadd	ST(0): +2.640000E+001

17.2.10 混合模式算术运算

到现在为止, 涉及到算术运算只包含实数, 应用程序经常涉及到混合算术运算: 同时包括整数和实数的运算。ADD 和 MUL 等整数算术运算指令不能处理实数, 因此惟一的选择是使用浮点指令。Intel 指令集中提供了提升整数至实数的指令以及把值加载至浮点栈的指令。

例子: 下面的 C++ 代码把一个整数和一个双精度数相加, 其和存储在一个双精度数中。在执行加法之前 C++ 自动把整数提升到实数:

```

int N = 20;
double X = 3.5;
double Z = N + X;

```

下面是等价的汇编语言代码:

```

.data
N SDWORD 20
X REAL8 3.5
Z REAL8 ?
.code
fld N                ; 加载整数至 ST(0) 中
fadd X               ; 内存操作数和 ST(0) 相加
fstp Z               ; 存储 ST(0) 至内存操作数中

```

例子: 下面的 C++ 程序把 N 提升成双精度数, 然后计算实数表达式的值, 最后再把结果存储到一个整数变量中:

```

int N = 20;
double X = 3.5;
int Z = (int) (N + X);

```

Visual C++ 生成的代码在 Z 中存储剪裁的结果之前调用了转换函数 `ftol`。如果以汇编语言编写实现该表达式的代码, 就可以使用 `FIST` 替代函数 `ftol`, Z 向上近似 (默认) 到 24:

```

fild N           ; 加载整数至 ST(0) 中
fadd X           ; 内存操作数和 ST(0) 相加
fist Z           ; 存储 ST(0) 至内存整数操作数中

```

改变近似模式：FPU 控制字的 RC 域允许指定近似的类型。可使用 FSTCW 把控制字存储到一个变量中，修改 RC 域（位 10 和位 11），然后再使用 FLDCW 指令把变量加载回控制字中：

```

fstcw ctrlWord   ; 存储控制字
or      ctrlWord,110000000000b ; 设置 RC = 剪裁方式
fldcw  ctrlWord   ; 加载控制字

```

对于前面的例子，如果使用剪裁的近似方法执行计算，得到的结果是 $Z = 23$ ：

```

fild N           ; 加载整数至 ST(0) 中
fadd X           ; 内存操作数和 ST(0) 相加
fist Z           ; 存储 ST(0) 至内存整数操作数中

```

此外，还可以重置近似模式至默认模式（近似到最近的偶数）：

```

fstcw ctrlWord   ; 存储控制字
and     ctrlWord,001111111111b ; 重置近似模式至默认
fldcw  ctrlWord   ; 加载控制字

```

17.2.11 屏蔽和未屏蔽的异常

浮点异常默认是屏蔽的（参见 17.2.3 节），因此在浮点异常发生时，处理器给结果赋一个默认值并继续安静地执行。例如，浮点数除 0 的结果是无穷大而不会终止程序：

```

.data
val1  DWORD 1
val2  REAL8 0.0
.code
fild  val1           ; 加载整数至 ST(0) 中
fdiv  val2           ; ST(0) = 正无穷大

```

如果在 FPU 控制字中未屏蔽异常，处理器将进入执行合适的异常处理程序。关闭异常屏蔽是通过清除 FPU 控制字中向合适的位完成的（参见表 17.17）。假如想要关闭对除零异常的屏蔽，下面是所需的步骤：

1. 存储 FPU 控制字至一个 16 位变量中。
2. 清除位 2（除零标志）。
3. 加载变量至控制字中。

下面的代码关闭对除零异常的屏蔽：

```

.data
ctrlWord WORD ?
.code
fstcw ctrlWord   ; 获取控制字
and     ctrlWord,111111111111011b ; 关闭对除零异常的屏蔽
fldcw  ctrlWord   ; 加载回 FPU 中

```

表 17.17 FPU 状态字中域

位	描 述
0	无效操作异常屏蔽位
1	反规格化操作数异常屏蔽位
2	除零异常屏蔽位
3	溢出异常屏蔽位

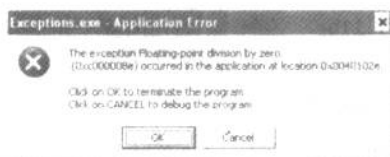
(续表)

位	描 述
4	下溢异常屏蔽位
5	精度异常屏蔽位
8~9	精度控制
10~11	近似控制
12	无穷大控制

现在, 如果执行下面的除零的代码, 就会产生一个未屏蔽的异常:

```
fild val1
fdiv val2          ; 除零
fst val2
```

FST 指令一开始执行, MS-Windows 就会显示下面的对话框:



屏蔽异常: 要屏蔽某种异常, 应设置 FPU 控制字中的相应位, 下面的代码屏蔽了除零异常:

```
.data
ctrlWord WORD ?
.code
fstcw ctrlWord          ; 获取控制字
or ctrlWord,100b        ; 屏蔽除零异常
fldcw ctrlWord          ; 加载回 FPU 中
```

17.2.12 本节习题

1. 写一条指令, 复制 ST(0)并加载至 FPU 堆栈。
2. 如果 ST(0)是在寄存器堆栈的 R6 寄存器中, 那么 ST(2)在哪个寄存器中?
3. 说出至少三个有特殊用途的 FPU 寄存器。
4. 如果一条浮点指令的第二个操作数是 B, 那么这表示该指令的操作数是什么类型?
5. 哪些指令接受立即数操作数?
6. FLD 允许的最大的数据类型是什么? 包含多少个数据位?
7. FSTP 指令与 FST 指令有什么不同?
8. 哪条浮点指令能够改变数字的符号?
9. FADD 指令都能使用哪些类型的操作数?
10. FISUB 指令与 FSUB 指令有什么不同?
11. 在 P6 系列之前的处理器上, 哪些指令用于比较两个浮点值?
12. 写两条指令, 把 FPU 的控制字送 EFLAGS 寄存器。
13. 哪条指令把一个整数操作数加载至 ST(0)中?
14. FPU 控制字中的哪个域允许改变 FPU 的近似模式?
15. 对于给定的精确结果 1.010101101, 使用 FPU 的默认近似模式把尾数近似到 8 位。
16. 对于给定的精确结果 -1.010101101, 使用 FPU 的默认近似模式把尾数近似到 8 位。

17. 写指令，实现下面的 C++ 代码：

```
double B = 7.8;
double M = 3.6;
double N = 7.1;
double P = -M * (N + B);
```

18. 写指令，实现下面的 C++ 代码：

```
int B = 7;
double N = 7.1;
double P = sqrt(N) + B;
```

17.3 Intel 指令编码

如果想要彻底理解汇编语言，读者需要花些时间研究汇编语言是如何翻译成机器语言的。由于 Intel 指令集中有大量的指令和寻址方式，因此这个主题相当复杂。本节首先以运行于实地址模式下的 8086/8088 微处理器作为示例，后面再介绍 Intel 引入 32 位处理器后发生的一些变化。

正如在第 2 章中所说的，Intel 8086 处理器是使用复杂指令集计算机（CISC）设计的第一个实现，该指令集中包含了多种内存寻址方式以及大量的移位指令、算术运算指令、数据传送指令、逻辑操作指令。与精简指令集计算机（RISC）的指令相比，Intel 指令的编码和解码需要一点技巧。编码一条指令的含义就是把汇编语言指令及其操作数转换成机器码，解码一条指令的含义是把指令的机器码转换成汇编语言。对于 Intel 指令编码和解码的学习，如果起不到别的作用的话，至少也会使你对 MASM 的作者多少有所感激！

17.3.1 IA-32 指令的格式

通用的 IA-32 机器指令格式（如图 17.6 所示）包含指令前缀，操作码，Mod R/M 字节，SIB（Scale Index Byte），地址偏移，立即数等部分。指令是以小尾顺序（little endian）存储的，因此前缀字节在指令的起始地址处，每条指令都有一个操作码，但其余的域就是可选的了，几乎没有指令包含所有这些域。平均而言，大多数指令都是 2 到 3 个字节长的。下面是各个域的简要描述：

- 指令前缀（instruction prefix）覆盖操作数的默认尺寸。
- 操作码（opcode）指明是哪条指令的哪个变量。例如，ADD 指令根据使用的参数类型的不同，有 9 种不同的操作码。

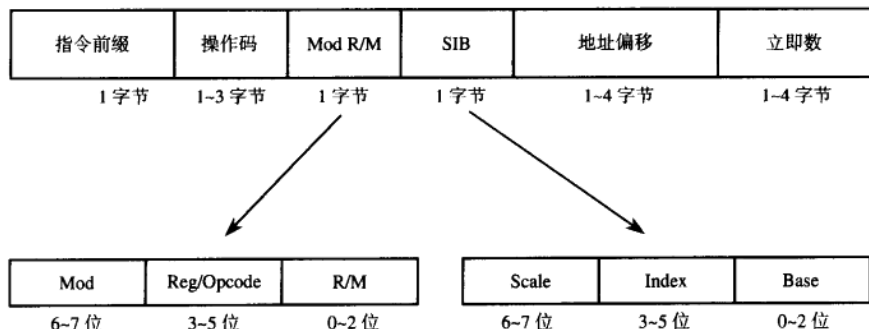


图 17.6 Intel IA-32 指令的格式

- Mod R/M 字节指明寻址模式和操作数。“R/M”代表寄存器（Register）和模式（Mode）。表 17.18 描述了 Mod 域不同取值的含义。表 17.19 描述了当 Mod = 10（二进制）时 R/M

域不同取值的含义。

- SIB 字节 (Scale Index Byte) 用于计算数组索引的偏移地址。
- 地址偏移 (Address Displacement) 域存放着操作数的偏移地址, 或者是可以在基址偏移寻址模式和基址变址寻址模式下加到基址寄存器或变址寄存器上的值。
- 立即数 (Immediate Data) 存放常量操作数。

表 17.18 Mod 域的值

Mod	含 义
00	无地址偏移部分 (除非 R/M=110)
01	无地址偏移的高半部分, 地址偏移的低半部分符号扩展至 16 位
10	地址偏移的高半部分和低半部分均有效
11	R/M 域是寄存器号

表 17.19 R/M 域的值

R/M	操 作 数
000	$[BX + SI] + D16^*$
001	$[BX + DI] + D16$
010	$[BP + SI] + D16$
011	$[BP + DI] + D16$
100	$[SI] + D16$
101	$[DI] + D16$
110	$[BP] + D16$
111	$[BX] + D16$

*D16 表示 16 位的地址偏移。

17.3.2 单字节指令

最简单的指令是无操作数或使用隐含操作数的指令, 这类指令仅需要操作码域, 操作码是由微处理器指令集预先定义好的。表 17.20 列出了一些常见的单字节指令。乍看起来, 好像 INC DX 指令是由于疏忽而被误放在表中了, 但事实上是 Intel 指令集的设计者决定为一些常用指令提供惟一的操作码, 因此它们对寄存器加一指令的代码大小和执行速度都进行了优化。

表 17.20 单字节指令

指 令	操 作 码
AAA	37
AAS	3F
CBW	98
LODSB	AC
XLAT	D7
INC DX	42

17.3.3 立即操作数送寄存器

立即数 (常量) 是以小尾顺序 (最低的字节最先存储) 存储在指令后面的。我们首先看看立

即数送寄存器的指令，先避开复杂的内存寻址模式。把立即数送寄存器的 MOV 指令的编码格式为 B8 +rw dw，其中操作码字节是 B8 +rw，+rw 表示 B8 要和一个寄存器号（0~7）相加，dw 表示后面跟一个立即字操作数（先低字节后高字节），操作码字节中可使用的寄存器编号如表 17.21 所示。下面例子中的数值都是十六进制的。

表 17.21 寄存器编号（8/16 位）

寄存器	编号
AX/SL	0
CX/CL	1
DX/DI	2
BX/BL	3
SP/AH	4
SI/DH	5
DI/BH	6
	7

例子：PUSH CX。对应的机器指令是 51，编码步骤如下：

1. 使用 16 位寄存器操作数的 PUSH 指令的操作码是 50。
2. CX 的寄存器号是 1，因此 1 加 50，得到机器码 51。

例子：MOV AX,1。对应的机器指令是 B8 01 00（十六进制）。编码步骤如下：

1. 立即数送 16 位寄存器的 MOV 指令的操作码是 B8。
2. AX 的寄存器编号是 0，因此 0 和 B8 相加（参见表 17.21）。
3. 立即数以小尾顺序（01 00）追加到指令的末尾。

例子：MOV BX,1234h。对应的机器指令是 BB 34 12。编码步骤如下：

1. 立即数送 16 位寄存器的 MOV 指令的操作码是 B8。
2. BX 的寄存器编号是 3，因此 3 和 B8 相加得到的操作码是 BB。
3. 立即操作数是字节 34 12。

在实践中，建议读者手动汇编几条立即数送寄存器的 MOV 指令以掌握其方法和要领，然后查看列表文件检查 MASM 生成的代码。

17.3.4 寄存器模式指令

在只使用寄存器操作数的指令中，Mod R/M 字节中分别使用三个数据位指定每个寄存器操作数。表 17.22 给出了寄存器的位编码。

表 17.22 确定 Mod R/M 域中的寄存器

R/M	寄存器	R/M	寄存器
000	AX 或 AL	100	SP 或 AH
001	CX 或 CL	101	BP 或 CH
010	DX 或 DL	110	SI 或 DH
011	BX 或 BL	111	DI 或 BH

例如, MOV AX, BX 的机器码是 89 D8。把寄存器数据送任何操作数的 16 位 MOV 指令的 Intel 编码是 89 /r, /r 表示操作码后跟一个 Mod R/M 字节。Mod R/M 字节由三个域组成 (mod, reg, R/M)。以 D8 为例, 包含下面三个域:

mod	reg	R/M
11	011	000

- 位 6~7 是 mod 域, 表示寻址方式, mod 域的值为 11, 指示 R/M 域存放着寄存器号。
- 位 3~5 是 reg 域, 表示源操作数, 在上例中, BX 的寄存器号是 011。
- 位 0~2 是 R/M 域, 表示目的操作数, 在上例中, AX 的寄存器号是 000。

表 17.23 给出了一些使用 8 位和 16 位寄存器操作数的例子。

表 17.23 MOV 指令编码和使用寄存器操作数的例子

指 令	操 作 码	mod	reg	R/M
mov ax,dx	8B	11	000	010
mov al,dl	8A	11	000	010
mov cx,dx	8B	11	001	010
mov cl,dl	8A	11	001	010

17.3.5 IA-32 的操作数尺寸前缀

下面我们把注意力转移到 32 位 Intel 处理器指令的编码上来。有些 IA-32 机器指令是以操作数尺寸前缀 (66h) 开头的, 该前缀修改了其指令的段属性。问题是, 为什么要用操作数尺寸指令前缀呢? 在实现 8088/8086 指令集时, 几乎所有的 256 个可能的操作码都被用于处理 8 位和 16 位的操作数了, 在 Intel 引入 32 位处理器时, 就必须使用一些新的操作码处理 32 位操作数, 同时与老处理器兼容。因此, 对于使用 16 位处理器的程序, 指令使用 16 位的操作数是默认的, 任何使用 32 位操作数的指令都要增加一个前缀字节。对于使用 32 位处理器的程序, 指令使用 32 位的操作数是默认的, 任何使用 16 位操作数的指令都要增加一个前缀字节。程序无论使用 32 位还是 16 位处理器, 对于使用 8 位操作数的指令, 无需使用任何前缀。

例子: 16 位操作数。下面通过汇编表 17.23 中给出的 MOV 指令来看看前缀字节在 16 位模式下是如何工作的。下例使用 .286 伪指令指示编译后代码的目标处理器, 以确保不会使用 32 位的寄存器。下面例子中每条 MOV 指令后都给出了相应的机器码:

```
.model small
.286
.stack 100h
.code
main PROC
    mov     ax,dx          ; 8B C2
    mov     al,dl          ; 8A C2
```

(这里没有使用 Irvine16.inc, 因为其面向的目标处理器是 386。)

接下来, 使用 .386 伪指令为 32 位处理器汇编同样的指令, 这时默认的操作数是 32 位的, 下面的例子中既使用了 16 位操作数也使用了 32 位操作数。第一条 MOV 指令无须前缀, 因为它使用的是 32 位操作数。第二条 MOV 指令由于使用的是 16 位的操作数, 因此需要使用操作数尺寸前缀:

```

.model small
.386
.stack 100h
.code
main PROC
    mov     eax,edx           ; 8B C2
    mov     ax,dx             ; 66 8B C2
    mov     al,dl             ; 8A C2

```

17.3.6 内存模式指令

如果 Mod R/M 字节只用于指示寄存器操作数的话, 那么 Intel 指令的编码将比事实上简单得多。事实上, Intel 汇编语言有大量的寻址方式, 导致 Mod R/M 字节的编码相当复杂 (IA-32 指令集的复杂性是许多精简指令集计算机设计的支持者对其批评的主要来源之一)。

该字节可指定 256 种不同的操作数组合, 表 17.24 列出了 Mod 域等于 00 时 Mod R/M 字节的编码 (十六进制) (完整的列表可在 *IA-32 Intel Architecture Software Developer's Manual* 卷 II 中找到)。表格中的 Mod 列包含两位, 用于指示寻址方式的分类, 例如在 Mod 等于 00 的情况下, 有 8 种可能的 R/M 值 (二进制 000 到 111), R/M 域用于指示有效地址列中的操作数类型。

例如, 要对指令 MOV [SI], AX 进行编码, 通过查表我们知道: Mod 域是 00, R/M 域是二进制值 100, 通过表 17.19 我们知道 AX 的寄存器号是二进制值 000, 因此完整的 Mod R/M 字节是二进制值 00 000 100, 也就是十六进制值 04。

mod	reg	R/M
00	000	100

十六进制值 04 在 AX 标记的列的第 5 行上。

MOV [SI], AL 的 Mod R/M 字节也是 04, 这时由于 AL 的寄存器号也是 000, 从 8 位寄存器移出数据的 MOV 指令的操作码是 88, Mod R/M 字节是 04, 因此机器指令是 88 04。

表 17.24 Mod R/M 字节部分列表 (16 位段)

字节		AL	CL	DL	BL	AH	CH	DH	BH	
字		AX	CX	DX	BX	SP	BP	SI	DI	
寄存器 ID		000	001	010	011	100	101	110	111	
Mod	R/M	Mod R/M 值								有效地址
00	000	00	08	10	18	20	28	30	38	[BX + SI]
	001	01	09	11	19	21	29	31	39	[BX + DI]
	010	02	0A	12	1A	22	2A	32	3A	[BP + SI]
	011	03	0B	13	1B	23	2B	33	3B	[BP + DI]
	100	04	0C	14	1C	24	2C	34	3C	[SI]
	101	05	0D	15	1D	25	2D	35	3D	[DI]
	110	06	0E	16	1E	26	2E	36	3E	16 位偏移
	111	07	0F	17	1F	27	2F	37	3F	[BX]

MOV 指令的例子

8 位和 16 位 MOV 指令的操作码和格式如表 17.25 所示。表 17.26 和表 17.27 解释了表 17.25 中用的缩略语。手动汇编 MOV 指令时可参考这些表格 (如果想了解更多这方面的细节, 请参考

IA-32 Intel Architecture Software Developer's Manual, 卷 II)。

表 17.25 MOV 指令的操作码

操作码	指 令	描 述
88 /r	MOV eb, rb	字节寄存器送有效地址处的字节
89 /r	MOV ew, rw	字寄存器送有效地址处的字
8A /r	MOV rb, eb	有效地址处的字节送字节寄存器
8B /r	MOV rw, ew	有效地址处的字送字寄存器
8C /0	MOV ew, ES	ES 送有效地址处的字
8C /1	MOV ew, CS	CS 送有效地址处的字
8C /2	MOV ew, SS	SS 送有效地址处的字
8C /3	MOV DS, ew	DS 送有效地址处的字
8E /0	MOV ES, mw	内存字送 ES
8E /0	MOV ES, rw	字寄存器送 ES
8E /2	MOV SS, mw	内存字送 SS
8E /2	MOV SS, rw	字寄存器送 SS
8E /3	MOV DS, mw	内存字送 DS
8E /3	MOV DS, rw	字寄存器送 DS
A0 dw	MOV AL, xb	(偏移 dw 处的) 字节变量送 AL
A1 dw	MOV AX, xw	(偏移 dw 处的) 字变量送 AX
A2 dw	MOV xb, AL	AL 送 (偏移 dw 处的) 字节变量
A3 dw	MOV xw, AX	AX 送 (偏移 dw 处的) 字变量
B0 +rb db	MOV rb, db	字节立即数送字节寄存器
B8 +rw dw	MOV rw, dw	字立即数送字寄存器
C6 /0 db	MOV eb, db	字节立即数送有效地址处的字节
C7 /0 dw	MOV ew, dw	字立即数送有效地址处的字

表 17.26 指令操作码符号说明

符 号	说 明
/n:	操作码之后紧跟一个 Mod R/M 字节, 其后还可能有立即数和偏移域。数字 n (0~7) 是 Mod R/M 字节的 reg 域的值
/r:	操作码之后紧跟一个 Mod R/M 字节, 其后还可能有立即数和偏移域
db:	操作码和 Mod R/M 字节之后有一个字节立即数
dw:	操作码和 Mod R/M 字节之后有一个字立即数
+rb:	8 位寄存器的寄存器代码 (0~7), 要和前面的十六进制格式字节相加形成 8 位的操作码
+rw:	16 位寄存器的寄存器代码 (0~7), 要和前面的十六进制格式的字节相加形成 8 位的操作码

表 17.27 指令操作数符号说明

符 号	说 明
db	-128~+127 之间的一个有符号值, 如果和字操作数联用, 该值进行符号扩展
dw	指令的字操作数
eb	一个字节操作数, 可以是寄存器操作数或内存操作数

(续表)

符 号	说 明
ew	一个字操作数，可以是寄存器操作数或内存操作数
rb	由值 0~7 标识的一个 8 位寄存器
rw	由值 0~7 标识的一个 16 位寄存器
xb	没有使用基址和变址寄存器的一个简单内存字节变量
xw	没有使用基址和变址寄存器的一个简单内存字变量

表 17.28 中包含了一些 MOV 指令的例子，读者可以手动汇编这些指令并把结果同表中对应的机器码比较。这里假设表中 myWord 变量开始于偏移 0102h 处。

表 17.28 一些 MOV 指令及其机器码示例

指 令	机 器 码	寻址方式
mov ax,myWord	A1 02 01	直接 (为 AX 进行了优化)
mov myWord,bx	89 1E 02 01	直接
mov [di],bx	89 1D	变址
mov [bx+2],ax	89 47 02	基址-偏移
mov [bx+si],ax	89 00	基址-变址
mov word ptr [bx+di+2],1234h	C7 41 02 34 12	相对基址变址

17.3.7 本节习题

1. 给出下面的 MOV 指令的操作码：

```
.data
myByte BYTE ?
myWord WORD ?
.code
mov ax,@data
mov ds,ax                ; a.
mov ax,bx                ; b.
mov bl,al                ; c.
mov al,[si]              ; d.
mov myByte,al            ; e.
mov myWord,ax            ; f.
```

2. 给出下面的 MOV 指令的操作码：

```
.data
myByte BYTE ?
myWord WORD ?
.code
mov ax,@data
mov ds,ax
mov es,ax                ; a.
mov dl,bl                ; b.
mov bl,[di]              ; c.
mov ax,[si+2]            ; d.
mov al,myByte            ; e.
mov dx,myWord            ; f.
```

3. 给出下面的 MOV 指令的 Mod R/M 字节：

```
.data
array WORD 5 DUP(?)
.code
```

```

mov ax,@data
mov ds,ax                ; a.
mov dl,b1                ; b.
mov bl,[di]              ; c.
mov ax,[si+2]            ; d.
mov ax,array[si]         ; e.
mov array[di],ax         ; f.

```

4. 给出下面的 MOV 指令的 Mod R/M 字节:

```

.data
array WORD 5 DUP(?)
.code
mov ax,@data
mov ds,ax
mov BYTE PTR array,5    ; a.
mov dx,[bp+5]           ; b.
mov [di],bx             ; c.
mov [di+2],dx           ; d.
mov array[si+2],ax      ; e.
mov array[bx+di],ax     ; f.

```

5. 手动汇编下面标记出来的指令并写出其十六进制的机器语言字节。假设 val1 位于偏移 0 处, 在使用 16 位值的地方, 值必须以小尾顺序出现:

```

.data
val1 BYTE 5
val2 WORD 256
.code
mov ax,@data
mov ds,ax                ; a.
mov al,val1              ; b.
mov cx,val2              ; c.
mov dx,OFFSET val1       ; d.
mov dl,2                 ; e.
mov bx,1000h             ; f.

```

17.4 本章小结

浮点数由三部分构成: 符号、尾数和指数。Intel 微处理器使用 IEEE 发布的 *Standard 754-1985 for Binary Floating-Point Arithmetic* 中规定的三种浮点二进制存储格式:

- 32 位的单精度值符号占 1 位、指数占 8 位、尾数中的小数部分占 23 位。
- 64 位的双精度值符号占 1 位、指数占 11 位、尾数中的小数部分占 52 位。
- 80 位的扩展精度值符号占 1 位、指数占 16 位、尾数中的小数部分占 63 位。

符号位为 1 表示是负数, 为 0 表示是正数。

浮点数的尾数部分由小数点及其前后数字构成。

在计算机中, 并非所有的 0 到 1 之间的实数都能用浮点数表示, 这是因为计算机只有有限个数据位。

正规化的有限数是所有 0 到 1 之间的可以用正规实数编码的 0 到无穷大之间的非零有限值。正无穷 ($+\infty$) 代表最大的正实数, 负无穷 ($-\infty$) 代表最小的负实数。NaN 是不能用有效浮点数表示的数。

Intel 8086 处理器是为处理整数运算而设计的, 因此 Intel 又引入了一种独立浮点协处理器芯片, 可以同 8086 一起插在主板上。从 Intel 486 处理器开始, 浮点处理硬件集成进了主 CPU, 称

为浮点单元 (FPU, Floating Point Unit)。

FPU 有 8 个可独立寻址的 8 位寄存器: R0~R7, 是以寄存器堆栈的形式组织的。在计算时, 浮点操作数是以扩展精度实数格式存放在 FPU 栈中的, 内存操作数也可用于浮点运算。FPU 在内存中保存算术运算的结果时, 自动把结果转换为以下格式之一: 整数、长整数、单精度 (短实数)、双精度 (长实数) 或压缩的二进制编码的十进制整数。

浮点指令总是以字母 F 开头, 以便与 CPU 指令区别开。指令的第二个字母 (通常是 B 或 I) 说明了内存操作数应如何解释: B 表示二/十进制 (BCD) 操作数, I 表示二进制整数操作数, 如果未指定 B 或 I, 就表示操作数是实数格式的。

Intel 8086 处理器是使用复杂指令集计算机 (CISC) 设计的第一个实现, 该指令集很大, 其中包含了多种内存寻址方式以及大量的移位指令、算术运算指令、数据传送指令、逻辑操作指令。

编码一条指令的含义就是把汇编语言指令及其操作数转换成机器码, 解码一条指令的含义是把指令的机器码转换成汇编语言。

IA-32 机器指令的格式包含指令前缀, 操作码, Mod R/M 字节, SIB (Scale Index Byte), 可选的立即数, 可选的地址偏移等部分。很少有指令包含所有这些域。前缀覆盖了目标处理器的默认操作数尺寸。操作码字节包含了指令唯一的操作代码。Mod R/M 字节指定了操作数及寻址方式。在仅使用寄存器操作数的指令中, Mod R/M 字节中分别使用三个数据位指定每个寄存器操作数。

17.5 编程练习

1. 浮点数的比较

以汇编语言重新实现下面 C++ 代码, 使用 WriteString 替换 printf() 函数调用:

```
double X;
double Y;
if( X < Y )
    printf("X is lower\n");
else
    printf("X is not lower\n");
```

(使用 Irvine32 库例程进行控制台输入输出, 不要调用标准 C 库的 printf 函数)。为 X 和 Y 赋一些不同的值测试编写的程序。

2. 以二进制形式显示浮点数

写一个程序, 接收一个单精度浮点二进制值, 以下面的格式显示该值: 显示+或-; 尾数 (前面是 1, 中间是二进制小数点, 后面是二进制尾数的小数部分); 指数 (以十进制显示为调整的指数值, 前面跟字母 E 以及指数的符号)。例如:

```
.data
sample REAL4 -1.75
```

输出如下:

```
-1.110000000000000000000000 E+0
```

3. 设置近似模式

(需要宏的知识。)写一个宏设置 FPU 的近似模式, 只有一个输入参数, 该参数是一个两个字

符的代码:

- RE: 近似到最接近的偶数
- RD: 向下近似趋向于 $-\infty$
- RU: 向上近似趋向于 $+\infty$
- RZ: 近似趋向于 0

宏的调用示例如下 (代码的大小写应无所谓):

```
mRound Re
mRound rd
mRound RU
mRound rZ
```

写一个小程序, 使用 FIST (存储整数) 指令测试各种可能的近似模式。

4. 表达式求值

写一个程序, 计算下面的表达式:

$((A + B) / C) * ((D - A) + E)$

给变量赋各种测试值并显示最后的结果。

5. 计算圆的面积

写一个程序, 提示用户输入圆的半径, 并计算圆的面积。使用本书附带链接库中的 ReadFloat 和 WriteFloat 过程, 使用 FLDPI 指令把 π 加载到浮点寄存器栈。

6. 边框绘制程序

提示用户输入二次多项式 $ax^2 + bx + c = 0$ 的系数 a, b, c。使用二次公式计算多项式的实数根。如果根是复数, 显示一条适当的信息。

7. 显示寄存器的状态值

标记寄存器 (参见 17.2.1 节) 使用两个数据位指定每个 FPU 寄存器内容的类型 (参见图 17.7)。可通过 FSTENV 指令加载标记字, 该指令填充如下的保护模式结构 (在 Irvine32.inc 中定义):

```
FPU_ENVIRON STRUCT
    controlWord    WORD ?
    ALIGN DWORD
    statusWord     WORD ?
    ALIGN DWORD
    tagWord        WORD ?
    ALIGN DWORD
    instrPointerOffset  DWORD ?
    instrPointerSelector  DWORD ?
    operandPointerOffset  DWORD ?
    operandPointerSelector WORD ?
    WORD ?          ; not used
FPU_ENVIRON ENDS
```

(在 Irvine16.inc 中定义了同名的结构用于实地址模式编程, 格式稍有不同。)

写一个程序, 在 FPU 栈上压入两个或更多的值, 调用 ShowFPUStack 显示堆栈, 显示每个 FPU 数据寄存器标记的值, 同时应显示与 ST(0) 对应的寄存器号 (使用 FSTSW 指令包状态字保存在一个 16 位的整数变量中, 然后从位 11 到 13 中析取出堆栈 TOP 指示器的值。)可参考下面的输出格式:

```

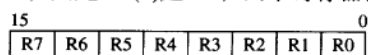
----- FPU Stack -----
ST(0): +1.5000000E+000
ST(1): +2.0000000E+000

R0 is empty
R1 is empty
R2 is empty
R3 is empty
R4 is empty
R5 is empty
R6 is valid
R7 is valid

ST(0) = R6

```

从输出示例中可以看出 ST(0)是 R6，因此 ST(1)是 R7，两个寄存器都包含了有效的浮点数。



TAG 的值:

00 = 有效

01 = 零

10 = 特殊值 (NaN、不支持的值、无穷大、反规格化值)

11 = 空

图 17.7 标记字的值

附录 A MASM 参考手册

本章要点

- 简介
- MASM 的保留关键字
- 寄存器名称
- Microsoft 汇编器 (ML)
- MASM 的伪指令
- 符号
- 操作符
- 运行时操作符

A.1 简介

Microsoft MASM 6.11 手册最后一次印刷是在 1992 年，它由三卷构成：

- 《程序员指南》
- 《参考手册》
- 《环境和工具》

遗憾的是，印刷的手册已经多年没有再版了，但是 Microsoft 在 Platform SDK 包中提供了手册的电子版 (MS-Word 文件)。现在，印刷的手册肯定早已是收藏家的收藏品了。

本章中的信息摘自《参考手册》的第 1 章至第 3 章，也包含了 MASM 6.14 readme.txt 文件中更新的内容。本书附带的 Microsoft 的版权许可允许读者拥有 MASM 的一份副本及其附带文档，其中文档的一部分已经在下面列出了。

格式符号：本附录使用一致的符号表示法，全部大写的单词表示 MASM 的保留字，保留字在程序中可能以大写形式也可能以小写形式出现。在下例中，DATA 是一个保留字：

```
.DATA
```

斜体的单词表示是已定义的术语或类型。在下例中，*number* 是指一个整数常量：

```
ALIGN [number]
```

在使用方括号括起一个项目的时候，表示项目是可选的。在下例中，*text* 是可选的：

```
[text]
```

在两个或多个项目之间出现垂直分隔符 (|) 时，必须选择其中的一个项目。下例表示要在 NEAR 和 FAR 之间进行选择：

```
NEAR|FAR
```

省略号 (...) 表示项目列表中的最后一个项目要进行重复。在下例中，*initializer* 之后的省略号表示它要多次重复：

```
[name] BYTE initializer [,initializer] ...
```

A.2 MASM 的保留关键字

\$	PARITY?
?	PASCAL
@ B	QWORD
@ F	REAL4
ADDR	REAL8
BASIC	REAL10
BYTE	SBYTE
C	SDWORD
CARRY?	SIGN?
DWORD	STDCALL
FAR	SWORD
FAR16	SYSCALL
FORTTRAN	TBYTE
FWORD	VARARG
NEAR	WORD
NEAR16	ZERO?
OVERFLOW?	

A.3 寄存器名称

AH	CR0	DR1	EBX	SI
AL	CR2	DR2	ECX	SP
AX	CR3	DR3	EDI	SS
BH	CS	DR6	EDX	ST
BL	CX	DR7	ES	TR3
BP	DH	DS	ESI	TR4
BX	DI	DX	ESP	TR5
CH	DL	EAX	FS	TR6
CL	DR0	EBP	GS	TR7

A.4 Microsoft 汇编器 (ML)

ML 程序 (ML.EXE) 可以用来汇编并链接一个或多个汇编语言源文件, 它的命令行选项是大小写敏感的, 格式为:

```
ML [options] filename [[options] filename] · · · [/link linkoptions]
```

命令行中要求至少有一个 *filename* 参数, 其中 *filename* 是汇编语言编写的源文件名。例如下面的命令行编译源文件 AddSub.asm 并生成目标文件 AddSub.obj:

```
ML -c AddSub.asm
```

options 参数由 0 到多个命令行选项构成, 每个选项以一个斜线 (/) 或减号 (-) 开始, 多个选项之间必须至少有一个空格分隔。表 A.1 列出了全部命令行选项, 命令行选项是大小写敏感的。

表 A.1 部分 ML 的命令行选项

选 项	含 义
/AT	允许支持微型内存模式。对与 .COM 文件格式的要求相冲突的代码给出错误信息。 注意该选项和 .MODEL TINY 伪指令并不完全相同
/B filename	选择其他的链接器
/c	只编译, 不链接
/coff	生成 Microsoft 公共目标文件格式 (common object file format) 的目标文件
/Cp	保留所有用户定义标识符的大小写
/Cu	映射所有标识符为大写形式
/Cx	保留公共和外部符号的大小写 (默认)
/Dsymbol[=value]	定义给定名字的文本宏。如果没有 <i>value</i> 部分, 文本宏为空。定义中以空格隔开的多个符号必须以引号引起来
/EP	生成一个预处理后的列表文件 (发送到 STDOUT)。参见 /Sf
/F hexnum	把堆栈大小设为 <i>hexnum</i> 个字节 (这与 /link /STACK: <i>number</i> 是相同的)。 <i>hexnum</i> 必须以十六进制格式表示。F 和 <i>hexnum</i> 之间必须有一个空格
/Fefilename	指定可执行文件名
/FI[filename]	生成一个汇编代码列表文件。参见 /Sf
/Fm[filename]	创建一个链接映像文件
/Fofilename	指定目标文件名
/FPi	为浮点运算生成模拟代码 (只用于混合语言编程)
/Fr[filename]	生成 .SBR 源浏览文件
/FR[filename]	生成扩展形式的 .SBR 源浏览文件
/Gc	指定使用 FORTRAN 或 Pascal 格式的函数调用约定和命名约定
/Gd	指定使用 C 格式的函数调用约定和命名约定
/Gz	使用 STDCALL 调用约定
/H number	外部名字限制为 <i>number</i> 个有效字符, 默认是 31 个字符
/help	调用 ML 的快速帮助
/I pathname	设置包含文件路径。最多允许 10 个 /I 选项
/link	链接器选项和库
/nologo	在编译成功的情况下屏蔽编译器输出的信息
/omf	产生 OMF (Microsoft Object Module Format) 文件。这种格式是老式 16 位 Microsoft 链接器 (LINK16.EXE) 所要求的
/Sa	打开所有可用信息列表
/Sc	在列表文件中增加指令执行时间信息
/Sf	在列表文件中增加第一遍编译后的列表信息
/Sg	使得 MASM 生成的代码出现在列表文件中。例如, 如果想要查看 .IF 和 .ELSE 伪指令如何工作, 则可以使用该选项
/Sl width	设置列表文件中行的宽度, 按每行字符数计算。范围在 60~255 之间, 或者为 0。默认情况下是 0, 同 PAGE width 伪指令

(续表)

选 项	含 义
<i>/Sn</i>	生成列表文件时关闭符号表
<i>/Sp length</i>	设置列表文件每页的长度, 按行数计算。范围是 10~255, 或者为 0, 默认情况下是 0。同 <i>PAGE length</i> 伪指令
<i>/Ss text</i>	为列表文件指定子标题, 同 <i>SUBTITLE text</i>
<i>/St text</i>	为列表文件指定标题, 同 <i>TITLE text</i>
<i>/Sx</i>	允许在列表文件中列出为假的条件块的清单
<i>/Ta filename</i>	汇编不以 .ASM 扩展名结尾的源文件
<i>/w</i>	同 <i>/W0</i>
<i>/Wlevel</i>	设置警告级别, <i>level</i> = 0, 1, 2, 3
<i>/WX</i>	视警告为错误
<i>/X</i>	忽略 INCLUDE 环境路径
<i>/Zd</i>	在目标文件中生成行号信息
<i>/Zf</i>	使所有符号变成公共符号
<i>/Zi</i>	在目标文件中生成 CodeView 需要的调试信息
<i>/Zm</i>	打开 M510 选项, 以最大程度地同 MASM 5.1 兼容
<i>/Zp[alignment]</i>	对结构按指定的字节边界对齐。 <i>alignment</i> 可以是 1, 2 或 4
<i>/Zs</i>	只进行参数检查
<i>/?</i>	显示 ML 命令行选项的帮助信息
<i>/error Report</i>	把汇编器内部的错误信息报告给 Microsoft

A.5 MASM 的伪指令

name = expression

把表达式 *expression* 的值赋给 *name*。这种符号后面可重复定义。

.186

允许汇编 80186 处理器指令, 禁用其后处理器引入的汇编指令。也允许汇编 8087 指令。

.286

允许汇编 80286 处理器的非特权指令, 禁用其后处理器引入的汇编指令。也允许汇编 80287 指令。

.286P

允许汇编 80286 处理器的所有指令 (包括特权指令), 禁用其后处理器引入的汇编指令。也允许汇编 80287 指令。

.287

允许汇编 80287 协处理器指令, 禁用其后协处理器引入的新处理器。

.386

允许汇编 80386 处理器的非特权指令, 禁用其后处理器引入的汇编指令。也允许汇编 80387 指令。

.386P

允许汇编 80386 处理器的所有指令 (包括特权指令), 禁用其后处理器引入的汇编指令。也允许汇编 80387 指令。

.387

允许汇编 80387 协处理指令。

.486

允许汇编 80486 处理器的非特权指令。

.486P

允许汇编 80486 处理器的所有指令 (包括特权指令)。

.586

允许汇编奔腾处理器的非特权指令。

.586P

允许汇编奔腾处理器的所有指令 (包括特权指令)。

.686

允许汇编奔腾 Pro 处理器的非特权指令。

.686P

允许汇编奔腾 Pro 处理器的所有指令 (包括特权指令)。

.8086

允许汇编 8086 指令 (以及等价的 8088 指令), 禁用其后处理器引入的指令。也允许汇编 8087 指令。这是处理器的默认模式。

.8087

允许汇编 8087 指令, 禁用其后处理器引入的指令。这是协处理器的默认模式。

ALIAS <alias> = <actual-name>

把旧函数名映射为新函数名。*alias* 是过程或函数的别名, *actual-name* 是函数或过程的实际名字, 尖括号是必需的。ALIAS 可用于创建库, 以使得链接器 (LINK) 把旧函数映射为新函数。

ALIGN [number]

使下一个变量或下一条指令按 *number* 对齐。

.ALPHA

使得段按字母顺序排序。

ASSUME segregister:name [, segregister:name] ···**ASSUME dataregister:type [, dataregister:type] ···****ASSUME register:ERROR [, register:ERROR] ···****ASSUME [register:] NOTHING [, register:NOTHING] ···**

允许对寄存器值的使用进行错误检查。在 ASSUME 生效之后, 汇编器监视给定寄存器值的改变是否符合类型约定。如果 ASSUME 中使用了 ERROR 关键字, 后面如果寄存器被使用, 汇编器就会报告错误。NOTHING 关键字禁止对寄存器做任何错误检查。可在一条语句中结合不同类型的 ASSUME。

.BREAK [.IF condition]

如果 *condition* 为真, 则退出.WHILE 或.REPEAT 块。

[name]BYTE initializer [, initializer]···

分配 1 字节存储空间并可选地为每个字节指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。

name CATSTR [textitem1 [, textitem2] ···]

连接文本。每个文本项都可以是一个字符串, 以 % 开头的常量或宏函数返回的字符串。

.CODE [name]

在和.MODEL 伪指令联用的时候, 表示名字是 *name* 的代码段的开始 (tiny, small, compact 和 flat 模式下默认代码段名是_TEXT, 其他模式下默认段名是 module_TEXT)。

COMM definition [definition] · · ·

定义公共变量, 每个变量及其属性在 *definition* 中定义, 每个 *definition* 的格式如下:

[*langtype*] [NEAR|FAR] *label*:*type* [:*count*]

label 是变量的名字。 *type* 可以是任何类型关键字 (BYTE, WORD 等) 或一个指定字节数的整数。 *count* 指定数据对象的数量 (默认为 1)。

COMMENT delimiter [text]

[*text*]

[*text*] *delimiter* [*text*]

把分隔符之间以及与分隔符在同一行上的文本作为注释。

.CONST

在和.MODEL 伪指令联用时, 表示常量数据段的开始 (段名是 CONST)。CONST 段的属性是只读的。

.CONTINUE [.IF condition]

如果条件 *condition* 为真, 跳转到.WHILE 或.REPEAT 块的开始。

.CREF

允许在符号表的符号部分和浏览文件中列出符号的交叉引用。

.DATA

在和.MODEL 伪指令联用时, 定义已初始化数据段的开始 (段名是_DATA)。

.DATA?

在和.MODEL 伪指令联用时, 定义未初始化数据段的开始 (段名是_BSS)。

.DOSSEG

根据 MS-DOS 的段约定组织段的顺序: 代码段最先, 然后是不在 DGROUP 段组中的段, 最后是 DGROUP 段组中的段。DGROUP 段组中的段顺序如下: 首先是非 BSS 和非 STACK 段, 然后是 BSS 段, 最后才是 STACK 段。主要用于确保 CodeView 支持 MASM 单独编译的程序。同 DOSSEG。

DOSSEG

同.DOSSEG, .DOSSEG 是推荐格式。

DB

用于和 BYTE 一样地定义数据。

DD

用于和 DWORD 一样地定义数据。

DF

用于和 FWORD 一样地定义数据。

DQ

用于和 QWORD 一样地定义数据。

DT

用于和 TBYTE 一样地定义数据。

DW

用于和 WORD 一样地定义数据。

[name]DWORD initializer [, initializer] · · ·

为双字（4 字节）分配存储并可选地为每个双字指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。

ECHO message

在标准输出设备（默认是屏幕）上显示消息 *message*。同 %OUT。

.ELSE

参见 .IF。

ELSE

标记条件块中可选块的开始。参见 IF。

ELSEIF

将 ELSE 和 IF 合并成一条伪指令。参见 IF。

ELSEIF2

如果 OPTION:SETIF2 为真，每当汇编时 ELSEIF 块都会被求值。

END [address]

标记模块的结束，还可以指定程序的入口点为 *address*。

.ENDIF

参见 .IF。

ENDIF

参见 IF。

ENDM

结束宏或重复块。参见 MACRO, FOR, FORC, REPEAT 或 WHILE。

name ENDP

标记前面以 PROC 开始的过程 *name* 的结束。参见 PROC。

name ENDS

标记前面以简化段伪指令及 SEGMENT, STRUCT, UNION 开始的段，结构或联合 *name* 的结束。

.ENDW

参见 .WHILE。

name EQU expression

把 *expression* 的值赋给 *name*。*name* 可以在后面重新定义。

name EQU <text>

将特定的文本 *text* 赋给 *name*。*name* 后面仍可以赋不同的文本值。参见 TEXTEQU。

.ERR [message]

产生一个错误信息。

.ERR2 [message]

如果 OPTION:SETIF2 为真，.ERR 块每当汇编时都会被求值。

.ERRB <textitem> [message]

如果 *textitem* 为空则产生一个错误。

.ERRDEF name [message]

如果 *name* 是已定义的标号、变量或符号就产生一个错误。

.ERRDIF[I]<textitem1>,<textitem2> [message]

如果 *textitem1* 和 *textitem2* 不相等就产生一个错误。如果使用了 I, 比较是大小写不敏感的。

.ERRE expression [message]

如果 *expression* 为假 (0) 则产生一个错误。

.ERRIDN[I]<textitem1>,<textitem2> [message]

如果 *textitem1* 和 *textitem2* 相等就产生一个错误。如果使用了 I, 比较是大小写不敏感的。

.ERRNB <textitem> [message]

如果 *textitem* 非空则产生一个错误。

.ERRNDEF name [message]

如果 *name* 是未定义的标号、变量或符号就产生一个错误。

.ERRNZ expression [message]

如果 *expression* 为真 (非 0) 则产生一个错误。

EVEN

把下一个变量或下一条指令对齐在偶数字节边界上。

.EXIT [expression]

生成结束码。可以选择向外壳程序返回值 *expression*。

EXITM [textitem]

结束当前的重复块或宏块的展开, 开始汇编块外的指令。在宏函数中, *textitem* 是返回值。

EXTERN [langtype] name[(altid)]:type [, langtype] name[(altid)]:type] · · ·

定义一个或多个名字为 *name* 并且类型为 *type* 的外部变量、标号或符号。类型 *type* 可以是 ABS, 此时把 *name* 作为常量引入。同 EXTRN。

EXTERNDEF [langtype] name:type [, langtype] name:type] · · ·

定义一个或多个名字为 *name* 并且类型为 *type* 的外部变量、标号或符号。如果 *name* 在模块中定义了, 就被作为 PUBLIC 符号对待; 如果 *name* 在模块中被引用, 被作为 EXTERN 声明的外部符号对待; 如果名字未被引用, 就被忽略掉。类型 *type* 可以是 ABS, 把 *name* 作为常量引入。该伪指令通常用于包含文件中。

EXTRN

参见 EXTERN。

.FARDATA [name]

在和 .MODEL 伪指令联用的时候, 定义初始化远程数据段的开始 (段名是 FAR_DATA 或 *name*)。

.FARDATA?[name]

在和 .MODEL 伪指令联用的时候, 定义未初始化远程数据段的开始 (段名是 FAR_BSS 或 *name*)。

FOR parameter [:REQ!=default], <argument [argument] · · · >

statements

ENDM

标记一个要为尖括号内每个参数 *argument* 重复一次的语句块, 每次重复的时候以当前 *argument* 替换 *parameter*。同 IRP。

FORC

parameter, <string> statements

ENDM

标记一个要为 *string* 中的每个字符重复一次的语句块, 每次重复的时候以 *string* 中的当前字符替换 *parameter*。同 **IRPC**。

[name] FWORD initializer [, initializer] . . .

分配 6 字节存储并可选地为每个 **FWORD** 指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。

GOTO macrolabel

将汇编控制转移到 *macrolabel* 标记的行。**GOTO** 只允许出现在 **MACRO**, **FOR**, **FORC**, **REPEAT** 和 **WHILE** 块中。标号必须单独成行, 而且必须以冒号开头。

name GROUP segment [, segment] . . .

把段 *segment* 添加到名为 *name* 的段组中。在 32 位平坦内存模式先使用该伪指令的时候无效果, 如果命令行指定了 */coff* 选项时会产生一个错误。

.IF condition1

statements

[**.ELSEIF** *condition2*

statements]

[**.ELSE**

statements]

.ENDIF

生成代码测试条件 *condition1* (如 $AX > 7$) 是否为真, 如果条件为真执行语句 *statements*。否则测试 *condition2* (如果有的话) 是否为真, 如果条件为真执行相应的语句 *statements*。如果最后跟 **.ELSE** 块, 那么在前面的条件都为假的情况下执行 **.ELSE** 后的语句块。注意条件是在运行时求值的。

IF expression 1

ifstatements

[**ELSEIF** *expression2*

elseifstatements]

[**ELSE**

elsestatements]

ENDIF

如果 *expression1* 为真 (非 0) 或 *expression1* 为假 (0), *expression2* 为真则确保汇编 *ifstatements* 或 *elseifstatements* 语句块。下面的伪指令可以代替 **ELSEIF**: **ELSEIFB**, **ELSEIFDEF**, **ELSEIFDIF**, **ELSEIFDIFI**, **ELSEIFE**, **ELSEIFIDN**, **ELSEIFIDNI**, **ELSEIFNB**, **ELSEIFNDEF**。如果前面的表达式都为假, 则汇编 *elsestatements* 语句块。注意表达式是在汇编时求值的。

IF2 expression

如果 **OPTION:SETIF2** 为真, 每当汇编时对 **IF** 块进行求值。完整的格式参见 **IF**。

IFB textitem

如果 *textitem* 为空则确保汇编其后的语句块。

IFDEF name

如果 *name* 是前面已经定义的标号、变量或符号时确保汇编其后的语句块。完整格式参见 **IF**。

IFDIF[I] *textitem1*, *textitem2*

如果 *textitem1* 和 *textitem2* 不同则确保汇编其后的语句块。如果使用了 I, 比较是大小写不敏感的。完整格式参见 IF。

IFE *expression*

如果 *expression* 为假, 确保汇编其后的语句块。完整格式参见 IF。

IFIDN[I] *textitem1*, *textitem2*

如果 *textitem1* 和 *textitem2* 相同则确保汇编其后的语句块。如果使用了 I, 比较是大小写不敏感的。完整格式参见 IF。

IFNB *textitem*

如果 *textitem* 非空则确保汇编其后的语句块。完整格式参见 IF。

IFNDEF *name*

如果 *name* 是未定义的标号、变量或符号时确保汇编其后的语句块。完整格式参见 IF。

INCLUDE *filename*

在汇编期间, 在当前源文件中插入 *filename* 源文件中的代码。如果 *filename* 包含了反斜线、分号、大于号、小于号、单引号或双引号时必须用尖括号括起。

INCLUDELIB *libraryname*

通知汇编器当前的模块应该和库 *libraryname* 相链接。如果 *libraryname* 包含了反斜线、分号、大于号、小于号、单引号或双引号时必须用尖括号括起。

name* INSTR [*position*], *textitem1*, *textitem2

查找 *textitem2* 在 *textitem1* 中首次出现的位置。开始位置 *position* 是可选的。每个文本项都可以是一个字符串、以 % 开头的常量或宏函数返回的字符串。

INVOKE *expression* [, *arguments*]

调用 *expression* 指定的地址处的过程, 根据语言类型隐含的调用约定在堆栈上或通过寄存器传递参数。每个传递给过程的参数都可以是一个表达式、一个寄存器或一个地址表达式 (前面跟 ADDR)。

IRP

参见 FOR。

IRPC

参见 FORC。

name* LABEL *type

创建一个标号 *name*, 赋以它当前地址指针计数器的值以及类型 *type*。

***name* LABEL [NEAR|FAR|PROC] PTR [*type*]**

创建一个标号 *name*, 赋以它当前地址指针计数器的值以及类型 *type*。

.K3D

允许汇编 K3D 指令。

.LALL

参见 LISTMACROALL。

.LFCOND

参见 LISTIF。

.LIST

列出语句。这是默认情况。

.LISTALL

列出所有的语句。等价于.LIST, .LISTIF 和.LISTMACROALL 的组合。

.LISTIF

列出假条件块内的所有语句。同.LFCOND。

.LISTMACRO

列出生成代码和数据的宏展开语句。这是默认的，同.XALL。

.LISTMACROALL

列出宏内的所有语句。同.LALL。

LOCAL *localname* [, *localname*]...

在宏内部定义对每个宏实例而言都是唯一的标号。

LOCAL *label* [[*count*]] [:*type*] [, *label* [[*count*]] [:*type*]]...

在过程定义中（PROC）创建在过程生存期内基于堆栈的变量。*label* 可以是一个简单的变量或包含 *count* 个元素的数组。

***name* MACRO [*parameter* [:REQ!:=default!;VARARG]]...**

statements

ENDM [*value*]

定义名为 *name* 的宏语句块，并为调用宏时传递的参数创建名为 *parameter* 的容器。宏函数向调用语句返回 *value*。

.MMX

允许汇编 MMX 指令。

.MODEL *memorymodel* [, *langtype*] [, *stackoption*]

初始化程序的内存模式。内存模式 *memorymodel* 可以是 TINY, SMALL, COMPACT, MEDIUM, LARGE, HUGE 或 FLAT。*langtype* 可以是 C, BASIC, FORTRAN, PASCAL, SYSCALL 或 STDCALL。*stackoption* 可以是 NEARSTACK 或 FARSTACK。

NAME *modulename*

忽略。

.NO87

禁止汇编所有的浮点指令。

.NOCREF [*name* [,*name*]...

禁止在符号表和浏览文件中列出符号。如果指定了符号名 *name*，那么只有给定名字的符号名被禁止。同.XCREF。

.NOLIST

禁止其后代码出现在程序的列表文件中。同.XLIST。

.NOLISTIF

禁止列出条件计算为假的条件块。这是默认的，同.SFCOND。

.NOLISTMACRO

禁止列出宏扩展。同.SALL。

OPTION *optionlist*

允许或禁止汇编器的某些特性。可用的选项包括：CASEMAP, DOTNAME, NODOTNAME, EMULATOR, NOEMULATOR, EPILOGUE, EXPR16, EXPR32, LANGUAGE, LJMP, NOLJMP, M510, NOM510, NOKEYWORD, NOSIGNEXTEND, OFFSET, OLDMACROS, NOOLDMACROS,

OLDSTRUCTS, NOOLDSTRUCTS, PROC, PROLOGUE, READONLY, NOREADONLY, SCOPED, NOSCOPE, SEGMENT, SETIF2。

ORG *expression*

把地址指针计数器设为 *expression* 的值。

%OUT

同 ECHO。

[*name*]OWORD *initializer* [, *initializer*] · · ·

分配 8 字 (16 字节) 存储并可选地为每个 OWORD 变量指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。该伪指令主要用于 SIMD 指令, 它能够容纳 4 个 4 字节实数。

PAGE [[*length*], *width*]

设置列表文件每页的行数和每行的字符宽度。如果不指定参数, 就生成一个分页符。

PAGE⁺

增加节号并把页号设为 1。

POPCONTEXT *context*

恢复全部或部分当前上下文 (由 PUSHCONTEXT 伪指令保存)。 *context* 可以是 ASSUMES, RADIX, LISTING, CPU 或 ALL。

***label* PROC [*distance*] [*langtype*] [*visibility*] [<*prologuearg*>]**

[USES *reglist*] [*parameter* [:*tag*]] · · ·

statements

***label* ENDP**

定义名为 *label* 的过程的开始和结束。PROC/ENDP 块内的语句可以使用 CALL 指令或 INVOKE 指令调用。

***label* PROTO [*distance*] [*langtype*] [, [*parameter*]:*tag*] · · ·**

声明过程原型。

PUBLIC [*langtype*] *name* [, [*langtype*] *name*] · · ·

使每个名为 *name* 的标号、变量和绝对符号对程序中的其他模块可用。

PURGE *macroname* [, *macroname*] · · ·

删除指定的宏定义。

PUSHCONTEXT *context*

保存当前上下文的全部或部分内容, 上下文包括: 段寄存器名、基数值、列表和交叉引用标志或处理器/协处理器的值。 *context* 可以是 ASSUMES, RADIX, LISTING, CPU 或 ALL。

[*name*]QWORD *initializer* [, *initializer*] · · ·

分配 8 字节存储并可选地为每个 QWORD 变量指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。

.RADIX *expression*

设置表达式默认的基数值为 *expression*, 其取值范围是 2~16。

***name* REAL4 *initializer* [, *initializer*] · · ·**

为单精度 (4 字节) 浮点数分配存储并可选地为每个单精度变量指定初始值 *initializer*。

***name* REAL8 *initializer* [, *initializer*] · · ·**

为双精度 (8 字节) 浮点数分配存储并可选地为每个双精度变量指定初始值 *initializer*。

name REAL10 initializer [, initializer] . . .

为 10 字节的浮点数分配存储并可选地为每个变量指定初始值 *initializer*。

recordname RECORD fieldname: width [= expression] [fieldname:width [= expression]] . . .

声明一个由指定域构成的记录。*fieldname* 指定域的名字, *width* 指定数据位的数目, *expression* 给出它的初始值。

.REPEAT

statements

.UNTIL condition

生成相应的代码重复执行 *statements* 语句块, 直到条件 *condition* 变为真。UNTILCXZ 可替代 UNTIL, 在 CX 为导致条件为真, 如果使用的是 UNTILCXZ, 其中的条件 *condition* 是可选的。

REPEAT expression

statements

ENDM

标记一个在汇编时由汇编器重复 *expression* 次的语句块。同 REPT。

REPT

参见 REPEAT。

.SALL

参见 .NOLISTMACRO。

name SBYTE initializer [, initializer] . . .

为有符号字节分配存储, 可选为其指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。

name SDWORD initializer [, initializer] . . .

为有符号双字 (4 字节) 分配存储, 可选为其指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。

name SEGMENT [READONLY] [align] [combine] [use] ['class']

statements

name ENDS

定义为 *name* 的段, 段属性有对齐 *align* (BYTE, WORD, DWORD, PARA, PAGE), 组合方式 *combine* (PUBLIC, STACK, COMMON, MEMORY, AT 地址, PRIVATE) 和模式 *use* (USE16, USE32 和 FLAT), 以及类别 *class*。

.SEQ

按顺序 (默认顺序) 排列段。

.SFCOND

参见 .NOLISTIF。

name SIZESTR textitem

获取文本项 *textitem* 的大小。

.STACK [size]

在和 .MODEL 联合使用的时候, 定义一个堆栈段 (段名为 STACK)。可选的 *size* 指定堆栈的字节数 (默认是 1024)。**.STACK** 伪指令自动结束堆栈段。

.STARTUP

生成程序启动代码。

STRUC

参见 STRUCT。

name STRUCT [*alignment*][,NONUNIQUE]

fielddeclarations

name ENDS

声明一个结构类型，每个域必须是一个有效的数据定义。同 STRUC。

name SUBSTR *textitem*,*position*, [, *length*]

返回 *textitem* 中从 *position* 开始的子串。*textitem* 可以是字符串、以%开始的字符串常量或宏函数返回的字符串。

SUBTITLE *text*

定义列表文件的子标题。同 SUBTTL。

SUBTTL

参见 SUBTITLE。

name SWORD *initializer* [, *initializer*] · · ·

为有符号字（2 字节）分配存储，可选为其指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。

[*name*] **TBYTE** *initializer* [, *initializer*] · · ·

分配 10 字节存储并可选地为其指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。

name TEXTEQU [*textitem*]

将文本 *textitem* 赋给 *name*。*textitem* 可以是字符串、以%开始的常量或宏函数返回的字符串。

.TFCOND

允许列出计算为假的条件块的清单。

TITLE *text*

定义程序列表文件的标题。

name TYPEDEF *type*

定义一个与类型 *type* 等价的名为 *name* 的新类型。

name UNION[*alignment*] [, NONUNIQUE]

fielddeclarations

[*name*] **ENDS**

声明一个或多个数据类型的联合。*fielddeclarations* 必须是有效的数据定义。在嵌套的 UNION 定义中要忽略 ENDS 前面的名字 *name*。

.UNTIL

参见 REPEAT。

.UNTILCXZ

参见 REPEAT。

.WHILE *condition*

statements

.ENDW

生成代码在 *condition* 条件为真时重复执行 *statements* 语句块。

WHILE *expression*

statements

ENDW

在 *expression* 为真时重复 *statements* 汇编语句块。

[*name*] **WORD** *initializer* [, *initializer*] ···

分配字存储并可选地为其指定初始值 *initializer*。在可以使用类型的地方也可以用做类型关键字。

.XALL

参见.LISTMACRO。

.XCREF

参见.NOCREF。

.XLIST

参见.NOLIST。

.XMM

允许汇编 SIMD 扩展指令。

A.6 预定义符号

\$

地址计数器的当前值。

?

在数据声明中, 汇编器分配存储但并不初始化。

@@:

定义只能在 *label1* 和 *label2* 之间被识别的代码标号, 其中 *label1* 是代码的开始或前一个 @@: 标号, *label2* 是代码的结尾或下一个 @@: 标号。参见 @B 和 @F。

@B

前一个 @@: 标号的地址。

@CatStr (*string1* [, *string2* ···])

连接一个或多个字符串的宏函数。返回一个字符串。

@code

代码段的名称 (文本宏)。

@CodeSize

TINY, SMALL, COMPACT 和 FLAT 模式下该值为 0, MEDIUM, LARGE 和 HUGE 模式下该值为 1。

@Cpu

指定处理器模式的位掩码。

@CurSeg

当前段的名字 (文本宏)。

@data

默认数据段组的名字。除了 FLAT 模式之外, 其余所有模式下该值都等于 DGROUP。FLAT 模式下该值等于 FLAT。

@DataSize

TINY, SMALL, COMPACT 和 FLAT 模式下该值为 0, MEDIUM, LARGE 模式下该值为 1,

HUGE 模式下该值为 2。

@Date

系统时间, 格式是 mm/dd/yy (文本宏)。

@Environ (envvar)

环境变量 *envvar* 的值(宏函数)。

@F

下一个@@: 标号的地址。

@ fardata

.FARDATA 伪指令定义的段名(文本宏)。

@fardata?

.FARDATA? 伪指令定义的段名(文本宏)。

@FileCur

当前文件的名称(文本宏)。

@FileName

正被汇编文件的基本名(不包含文件的扩展名, 文本宏)。

@InStr ([position], string1, string2)

找出 *string2* 在 *string1* 中从 *position* 开始首次出现位置的宏函数, 如果没有 *position*, 从 *string1* 的开始查找, 如果未找到 *string2* 返回 0。

@Interface

关于语言参数的信息。

@Line

当前文件中的源代码行号。

@Model

TINY 模式该值为 1, SAML 模式该值为 2, COMPACT 模式该值为 3, MEDIUM 模式该值为 4, LARGE 模式该值为 5, HUGE 模式该值为 6, FLAT 模式该值为 7。

@SizeStr(string)

返回给定字符串长度的宏函数。返回一个整数。

@stack

对于近 (NEAR) 堆栈该值是 DGROUP, 对于远 (FAR) 堆栈该值是 STACK (文本宏)。

@SubStr (string, position[, length])

返回 *string* 中从 *position* 位置开始长度为 *length* 的子串。

@Time

24 小时格式的系统时间, 格式是 hh:mm:ss (文本宏)。

@Version

版本号, MASM 6.1 该值为 610 (文本宏)。

@WordSize

16 位段该值为 2, 32 位段该值为 4。

A.7 操作符

expression1 + expression2

返回 *expression1* 加 *expression2* 的和。

expression1 - expression2

返回 *expression1* 减 *expression2* 的差。

expression1 * expression2

返回 *expression1* 乘以 *expression2* 的乘积。

expression1 / expression2

返回 *expression1* 除以 *expression2* 的商。

-expression

expression 符号取反。

expression1[expression2]

返回 *expression1* 加 *expression2* 的和。

segment:expression

以 *segment* 覆盖 *expression* 的默认段。*segment* 可以是段寄存器、组名、段名或段表达式。*expression* 必须是常量。

expression.field[,field] . . .

返回 *expression* 的地址值加上结构或联合内 *field* 偏移的和。

[register].field[,field] . . .

返回由 *register* 指向的地址值加上结构或联合内 *field* 偏移的和。

<text>

把 *text* 作为文本对待。

"text"

把 *text* 作为字符串对待。

'text'

把 *text* 作为字符串对待。

!character

把 *character* 作为字符而不是操作符或符号对待。

;text

把 *text* 作为注释对待。

;;text

text 只在宏内以注释的形式出现。列表文件中在宏展开的时候并不显示 *text*。

%expression

把宏内的 *expression* 值作为文本对待。

¶meter&

以对应的参数值替换 *parameter*。

ABS

参见 EXTERNDEF 伪指令。

ADDR

参见 INVOKE 伪指令。

expression1 AND expression2

返回 *expression1* 和 *expression2* 进行与操作的结果。

count DUP (initialvalue[,initialvalue] . . .)

指定 *count* 数量的 *initialvalue* 的声明。

expression1 EQ expression2

如果 *expression1* 等于 *expression2* 返回真(-1), 否则返回假(0)。

expression1 GE expression2

如果 *expression1* 大于等于 *expression2* 返回真(-1), 否则返回假(0)。

expression1 GT expression2

如果 *expression1* 大于 *expression2* 返回真(-1), 否则返回假(0)。

HIGH expression

返回 *expression* 的高字节。

HIGHWORD expression

返回 *expression* 的高字。

expression1 LE expression2

如果 *expression1* 小于等于 *expression2* 返回真(-1), 否则返回假(0)。

LENGTH variable

返回 *variable* 内第一个初始化项占用的字节数。

LENGTHOF variable

返回 *variable* 中的数据对象的数目。

LOW expression

返回 *expression* 的低字节。

LOWWORD expression

返回 *expression* 的低字。

LROFFSET expression

返回 *expression* 的偏移。同 **OFFSET**, 不过它生成一个由加载器解析的偏移, 以允许重定位代码段。

expression1 LT expression2

如果 *expression1* 小于 *expression2* 返回真(-1), 否则返回假(0)。

MASK {recordfilename|record}

返回一个掩码, 其中 *recordfilename* 或 *record* 对应的位设置而其他位清除。

expression1 MOD expression2

返回 *expression1* 除以 *expression2* 时的余数值(模)。

expression1 NE expression2

如果 *expression1* 不等于 *expression2* 返回真(-1), 否则返回假(0)。

NOT expression

expression 的所有位变反。

OFFSET expression

返回 *expression* 的偏移地址。

OPATTR expression

返回一个定义 *expression* 模式和范围的字。低字节等于 **.TYPE** 返回的字节, 高字节包含了其他信息。

expression1 OR expression2

返回 *expression1* 和 *expression2* 进行或运算的结果。

type PTR expression

强制使表达式按 *type* 类型的值对待。

[distance] PTR type

指定一个指针的类型为 *type*。

SEG expression

返回 *expression* 所在的段。

expression SHL count

返回表达式 *expression* 左移 *count* 个位后的结果。

SHORT label

把 *label* 的类型设为短类型。所有跳转到 *label* 的跳转都必须是近跳转 (从跳转指令到 *label* 的距离在-128~+127 字节范围之内)。

expression SHR count

返回表达式 *expression* 右移 *count* 个位后的结果。

SIZE variable

返回初始化时为 *variable* 第一个初始化值分配的字节数。

SIZEOF {variable|type}

返回 *variable* 或 *type* 占用的字节数。

THIS type

返回一个指定的 *type* 类型的操作数,其偏移和段值与当前地址计数器的偏移地址和段值相同。

.TYPE expression

参见 OPATTR。

TYPE expression

返回 *expression* 的类型。

WIDTH {recordfilename|record}

返回当前 *recordfilename* 或 *record* 内数据位的数目。

expression1 XOR expression2

返回 *expression1* 和 *expression2* 进行位异或运算的结果。

A.8 运行时操作符

下面的操作符只能在 .IF, .WHILE 或 .REPEAT 块中使用,相关的表达式是在运行时进行求值,而不是在汇编时求值的。

expression1 == expression2

等于。

expression1 != expression2

不等于。

expression1 > expression2

大于。

expression1 >= expression2

大于等于。

expression1 < expression2

小于。

expression1 <= expression2

小于等于。

expression1 || expression2

逻辑或。

expression1 && expression2

逻辑与。

expression1 & expression2

位与。

!expression

逻辑非。

CARRY?

进位标志的状态值。

OVERFLOW?

溢出标志的状态值。

PARITY?

奇偶标志的状态值。

SIGN?

符号标志的状态值。

ZERO?

零标志的状态值。

附录 B IA-32 指令集

本章要点

- B.1 简介
- B.2 指令集（非浮点指令）
- B.3 浮点指令

B.1 简介

本附录是最常用的 IA-32 指令的速查手册，未包含系统模式指令或那些仅在操作系统内核代码或设备驱动中使用的指令。

B.1.1 标志

每条指令的说明中都包含一系列用于描述指令如何影响 CPU 状态标志的方格，其中的每个标志用一个字母表示：

O	溢出标志	S	符号标志	P	奇偶标志
D	方向标志	Z	零标志	C	进位标志
I	中断标志	A	辅助进位标志		

在方格中，使用下面的符号来表示每条指令是以何种方式影响标志的：

1	标志置位
0	标志清零
?	标志值的改变无法预测
空白	标志值不受影响
*	根据与该标志相关的特定规则修改标志的值

例如，下面的 CPU 标志图摘自某条指令的描述：

O	D	I	S	Z	A	P	C
?			?	?	*	?	*

从图中可见：溢出标志、符号标志、零标志和奇偶标志将改变为未知值，辅助进位标志和进位标志将根据与这些标志值相关的规则进行修改，方向标志和中断标志不会改变。

B.1.2 指令描述及指令格式

在引用源操作数和目的操作数时，我们使用 Intel 80x86 指令的自然顺序，即其中第一个操作数是目的操作数，第二个操作数是源操作数。例如在 MOV 指令中，目的操作数将被赋以源操作数的一份副本：

MOV 目的，源

一条指令可能有多种格式，表 B.1 是在指令格式描述中使用的符号的列表。在单条指令描述中，使用符号“(IA-32)”表示一条指令或其变体只能用于 IA-32 系列处理器（Intel 386 以上）。与

之类似, 符号“(80286)”表示至少要使用 80286 处理器。

寄存器符号如(E)CX, (E)SI, (E)DI, (E)SP, (E)BP 和(E)IP 等分别用于使用 32 位寄存器的 IA-32 处理器和使用 16 位寄存器的早期处理器。

表 B.1 指令格式中使用的符号

符 号	描 述
<i>reg</i>	下列 8 位、16 位或 32 位通用寄存器中的一个: AH, AL, BH, BL, CH, CL, DH, DL, AX, BX, CX, DX, SI, DI, BP, SP, EAX, EBX, ECX, EDX, ESI, EDI, EBP 和 ESP
<i>reg8, reg16, reg32</i>	通用寄存器, 数字后缀表示包含的数据位的位数
<i>segreg</i>	16 位段寄存器 (CS, DS, ES, SS, FS, GS)
<i>accum</i>	AL, AX 或 EAX
<i>mem</i>	使用任何标准内存寻址方式的内存操作数
<i>mem8, mem16, mem32</i>	内存操作数, 数字后缀表示操作数的位数
<i>shortlabel</i>	代码段内距当前位置-128~+127 字节范围之内的地址
<i>nearlabel</i>	当前代码段内的位置, 以标号标识
<i>farlabel</i>	外部代码段内的位置, 以标号标识
<i>imm</i>	立即操作数
<i>imm8, imm16, imm32</i>	立即操作数, 数字后缀表示操作数的位数
<i>instruction</i>	一条 80x86 汇编语言指令

B.2 指令集 (非浮点指令)

AAA	<p>加法后进行 ASCII 调整 (ASCII Adjust After Addition)</p> <table border="1"><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>?</td><td></td><td></td><td>?</td><td>?</td><td>*</td><td>?</td><td>*</td></tr></table> <p>调整两个 ASCII 数字相加之后在 AL 中的结果。如果 AL 的低 4 位大于 9 或辅助进位标志等于 1, 则 AL 加 6 并清除 AL 的高 4 位, 同时 AH 加 1, 设置进位标志和辅助进位标志; 否则, 直接清除 AL 的高 4 位, 清除进位标志和辅助进位标志。</p> <p>指令格式:</p> <p>AAA</p>	O	D	I	S	Z	A	P	C	?			?	?	*	?	*
O	D	I	S	Z	A	P	C										
?			?	?	*	?	*										

AAD	<p>在除法前进行 ASCII 调整 (ASCII Adjust Before Division)</p> <table border="1"><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>?</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>?</td></tr></table> <p>将 AH 和 AL 中未压缩的 BCD 数字转换成二进制数值存放在 AL 中, AH 清零, 为 DIV 指令做好准备。</p> <p>指令格式:</p> <p>AAD</p>	O	D	I	S	Z	A	P	C	?			*	*	?	*	?
O	D	I	S	Z	A	P	C										
?			*	*	?	*	?										

AAM	乘法后进行 ASCII 调整 (ASCII Adjust After Multiply)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>?</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>?</td></tr></table>	O	D	I	S	Z	A	P	C	?			*	*	?	*	?
	O	D	I	S	Z	A	P	C									
?			*	*	?	*	?										
调整两个未压缩的 BCD 数字相乘之后 AX 中的结果, 即将二进制值调整为非压缩的 ASCII 格式。调整方法是 AL 除以 0Ah, 得到的商存放在 AH 中, 余数存放在 AL 中。 指令格式: AAM																	

AAS	减法后进行 ASCII 调整 (ASCII Adjust After Subtraction)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>?</td><td></td><td></td><td>?</td><td>?</td><td>*</td><td>?</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	?			?	?	*	?	*
	O	D	I	S	Z	A	P	C									
?			?	?	*	?	*										
<p>调整 ASCII 减法之后 AL 中得到的结果。如果 AL 的低 4 位大于 9 或辅助进位标志等于 1, AL 减 6, 清除 AL 的高 4 位, AH 减 1, 设置进位标志和辅助进位标志。否则, 直接清除 AL 的高 4 位, 清除进位标志和辅助进位标志。</p> <p>指令格式:</p> <p style="text-align: center;">AAS</p>																	

ADC	带进位加 (Add Carry)																	
		<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	*	*	*
	O	D	I	S	Z	A	P	C										
*			*	*	*	*	*											
将源操作数、目的操作数和进位标志相加。操作数尺寸必须相同。 指令格式： <table><tr><td>ADC</td><td>reg, reg</td><td>ADC</td><td>reg, imm</td></tr><tr><td>ADC</td><td>mem, reg</td><td>ADC</td><td>mem, imm</td></tr><tr><td>ADC</td><td>reg, mem</td><td>ADC</td><td>accum, imm</td></tr></table>		ADC	reg, reg	ADC	reg, imm	ADC	mem, reg	ADC	mem, imm	ADC	reg, mem	ADC	accum, imm					
ADC	reg, reg	ADC	reg, imm															
ADC	mem, reg	ADC	mem, imm															
ADC	reg, mem	ADC	accum, imm															

ADD	加 (Add)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	*	*	*
	O	D	I	S	Z	A	P	C									
*			*	*	*	*	*										
<p>源操作数与目的操作数相加，结果存储在目的操作数中。操作数尺寸必须相同。</p> <p>指令格式：</p> <table><tr><td>ADD</td><td>reg, reg</td><td>ADD</td><td>reg, imm</td></tr><tr><td>ADD</td><td>mem, reg</td><td>ADD</td><td>mem, imm</td></tr><tr><td>ADD</td><td>reg, mem</td><td>ADD</td><td>accum, imm</td></tr></table>	ADD	reg, reg	ADD	reg, imm	ADD	mem, reg	ADD	mem, imm	ADD	reg, mem	ADD	accum, imm					
ADD	reg, reg	ADD	reg, imm														
ADD	mem, reg	ADD	mem, imm														
ADD	reg, mem	ADD	accum, imm														

AND	逻辑与 (Logical AND)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>0</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>0</td></tr></table>	O	D	I	S	Z	A	P	C	0			*	*	?	*	0
	O	D	I	S	Z	A	P	C									
0			*	*	?	*	0										
<p>目的操作数中的每个数据位与源操作数中的对应位进行与操作。</p> <p>指令格式：</p> <table><tr><td>AND</td><td>reg, reg</td><td>ADD</td><td>reg, imm</td></tr><tr><td>AND</td><td>mem, reg</td><td>ADD</td><td>mem, imm</td></tr><tr><td>AND</td><td>reg, mem</td><td>ADD</td><td>accum, imm</td></tr></table>	AND	reg, reg	ADD	reg, imm	AND	mem, reg	ADD	mem, imm	AND	reg, mem	ADD	accum, imm					
AND	reg, reg	ADD	reg, imm														
AND	mem, reg	ADD	mem, imm														
AND	reg, mem	ADD	accum, imm														

BOUND	检查数组边界 (Check Array Bounds) (80286)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C								
	O	D	I	S	Z	A	P	C									
<p>检查一个有符号的指针值是否在数组边界之内。在 80286 处理器上, 目的操作数可以是任何包含要检查的指针的 16 位寄存器。源操作数必须是 32 位内存操作数, 其高字和低字分别包含数组的上边界指针值和下边界指针值。在 IA-32 上, 目的操作数也可以是 32 位寄存器, 源操作数可以是 64 位内存操作数。</p> <p>指令格式:</p> <table><tr><td>BOUND</td><td>reg16, mem32</td><td>BOUND</td><td>reg32, mem64</td></tr></table>	BOUND	reg16, mem32	BOUND	reg32, mem64													
BOUND	reg16, mem32	BOUND	reg32, mem64														

BSF BSR	位扫描 (Bit Scan) (IA-32)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>?</td><td></td><td></td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table>	O	D	I	S	Z	A	P	C	?			?	?	?	?
O	D	I	S	Z	A	P	C									
?			?	?	?	?	?									
<p>扫描操作数, 寻找第一个被设置的数据位。如果找到则清除零标志, 目的操作数存放第一个被设置位的位号 (索引)。如果没有找到被设置的数据位则 ZF=1。BSF 指令按照从位 0 到最高位的顺序扫描, BSR 指令按从最高位到位 0 的顺序扫描。</p> <p>指令格式 (以 BSF 为例, 适用于 BSF 和 BSR, 后面有些指令的描述方法类似, 不再重复说明):</p> <table><tr><td>BSF</td><td>reg16, r/m16</td><td>BSF</td><td>reg32, r/m32</td></tr></table>		BSF	reg16, r/m16	BSF	reg32, r/m32											
BSF	reg16, r/m16	BSF	reg32, r/m32													

BSWAP	字节交换 (Byte Swap) (IA-32)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C								
	O	D	I	S	Z	A	P	C									
反转 32 位目的寄存器中的字节顺序。 指令格式： BSWAP reg32																	

BT BTC BTR BTS	位测试 (Bit Tests) (IA-32)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>?</td><td></td><td></td><td>?</td><td>?</td><td>?</td><td>?</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	?			?	?	?	?	*
O	D	I	S	Z	A	P	C										
?			?	?	?	?	*										
	<p>将指定位 (n) 复制到进位标志中, 目的操作数包含要操作的位号。BT 将源操作数的位 n 复制到进位标志中; BTC 将源操作数的位 n 复制到进位标志中并将位 n 变反; BTR 将源操作数的位 n 复制到进位标志中并将位 n 清除; BTS 将源操作数的位 n 复制到进位标志中并将位 n 置 1。</p> <p>指令格式 (以 BT 为例):</p> <table><tr><td>BT $r/m16, imm8$</td><td>BT $r/m16, r16$</td></tr><tr><td>BT $r/m32, imm8$</td><td>BT $r/m32, r32$</td></tr></table>	BT $r/m16, imm8$	BT $r/m16, r16$	BT $r/m32, imm8$	BT $r/m32, r32$												
BT $r/m16, imm8$	BT $r/m16, r16$																
BT $r/m32, imm8$	BT $r/m32, r32$																

CALL	调用过程 (Call a Procedure)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C								
	O	D	I	S	Z	A	P	C									
<p>将下一条指令的地址压入堆栈并将控制转移到目的地址。如果过程是近过程（在同一个段内），指令只压入下一条指令的偏移，否则下一条指令的段和偏移都被压入堆栈。</p> <p>指令格式：</p> <table><tr><td>CALL</td><td><i>nearlabel</i></td><td>CALL</td><td><i>mem16</i></td></tr><tr><td>CALL</td><td><i>farlabel</i></td><td>CALL</td><td><i>mem32</i></td></tr><tr><td>CALL</td><td><i>reg</i></td><td></td><td></td></tr></table>	CALL	<i>nearlabel</i>	CALL	<i>mem16</i>	CALL	<i>farlabel</i>	CALL	<i>mem32</i>	CALL	<i>reg</i>							
CALL	<i>nearlabel</i>	CALL	<i>mem16</i>														
CALL	<i>farlabel</i>	CALL	<i>mem32</i>														
CALL	<i>reg</i>																

CBW	字节扩展到字 (Convert Byte to Word)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C							
O	D	I	S	Z	A	P	C									
	将 AL 中的符号位扩展到 AH 中。															
	指令格式:															
	CBW															

CDQ	双字扩展到 8 字节 (Convert Doubleword to Quadword) (IA-32)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C							
O	D	I	S	Z	A	P	C									
	将 EAX 中的符号位扩展到 EDX 中。															
	指令格式:															
	CDQ															

CLC	清除进位标志 (Clear Carry Flag)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>0</td></tr></table>	O	D	I	S	Z	A	P	C							
O	D	I	S	Z	A	P	C									
							0									
	清除进位标志。 指令格式： CLC															

CLD	清除方向标志 (Clear Direction Flag)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C		0						
	O	D	I	S	Z	A	P	C									
	0																
清除方向标志。此时，字符串指令自动增加(E)SI 和(E)DI 的值。 指令格式： CLD																	

CLI	清除中断标志 (Clear Interrupt Flag)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td>0</td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C			0					
	O	D	I	S	Z	A	P	C									
		0															
清除中断标志。禁止可屏蔽硬件中断，直到执行一条 STI 指令为止。 指令格式： CLI																	

CMC	进位标志取反 (Complement Carry Flag)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C								*
	O	D	I	S	Z	A	P	C									
							*										
当前进位标志值变反。 指令格式： CMC																	

CMP	比较 (Compare)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	*	*
O	D	I	S	Z	A	P	C									
*			*	*	*	*	*									
比较目的操作数和源操作数, 隐含执行 (相应设置标志位, 但不改变操作数) 从源操作数中 减掉目的操作数的减法操作。																
指令格式:																
CMP reg, reg	CMP reg, imm															
CMP mem, reg	CMP mem, imm															
CMP reg, mem	CMP accum, imm															

CMPBS	比较字符串 (Compare Strings)																
CMPSB																	
CMPSW	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	*	*	*
O	D	I	S	Z	A	P	C										
*			*	*	*	*	*										
CMPSD	<p>比较内存中由 DS:(E)SI 和 ES:(E)DI 寻址的字符串。隐含执行源操作数减去目的操作数的减法操作。CMPSB 比较字节, CMPSW 比较字, CMPSD 比较双字 (IA-32 处理器)。(E)SI 和 (E)DI 的值依据操作数的大小以及方向标志的状态增减。如果设置了方向标志, 减少 (E)SI 和 (E)DI, 否则增加 (E)SI 和 (E)DI。</p> <p>指令格式:</p> <table><tr><td>CMPSB</td><td>CMPSW</td></tr><tr><td>CMPSD</td><td></td></tr></table>	CMPSB	CMPSW	CMPSD													
CMPSB	CMPSW																
CMPSD																	

CMPXCHG	比较并交换(Compare and Exchange)								
	<div>O D I S Z A P C</div> <div><table><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table></div>	*			*	*	*	*	*
	*			*	*	*	*	*	
目的操作数和累加器 (AL, AX 或 EAX) 进行比较, 如果相等, 源操作数复制到目的操作数中, 否则目的操作数复制到累加器中。									

指令格式:

CMPXCHG *reg, reg* CMPXCHG *mem, reg*

CWD	字扩展到双字 (Convert Word to Double Word)								
	<div><div>O D I S Z A P C</div><table><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></div>								
AX 符号位扩展到 DX。 指令格式： CWD									

DAA	加法后进行十进制调整 (Decimal Adjust After Addition)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>?</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	?			*	*	*	*	*
	O	D	I	S	Z	A	P	C									
?			*	*	*	*	*										
调整两个压缩 BCD 值相加之后 AL 中的结果。将和转换成两个 BCD 数存放在 AL 中。 指令格式： DAA																	

DAS	减法后进行十进制调整 (Decimal Adjust After Subtraction)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>?</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	?			*	*	*	*	*
	O	D	I	S	Z	A	P	C									
?			*	*	*	*	*										
将两个压缩 BCD 值减法操作后在 AL 中得到的结果转换为两个压缩的 BCD 数并存储在 AL 中。 指令格式： DAS																	

DEC	<p>减 1 (Decrement)</p> <table border="1"><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*				*	*	*	*
O	D	I	S	Z	A	P	C										
*				*	*	*	*										
	<p>从操作数中减去 1。本指令不影响进位标志。</p> <p>指令格式：</p> <p>DEC reg </p>																

IMUL	<p>有符号整数乘法 (Signed Integer Multiply)</p> <table><tr><td></td><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td></td><td>?</td><td>?</td><td>?</td><td>?</td><td>*</td></tr></table> <p>对 AL、AX 或 EAX 执行有符号整数乘法操作。如果乘数是 8 位的，被乘数在 AL 中，积在 AX 中；如果乘数是 16 位的，被乘数在 AX 中，积在 DX:AX 中；如果乘数是 32 位的，被乘数在 EAX 中，积在 EDX:EAX 中。如果 16 位的积扩展到 AH、32 位的积扩展到 DX 或 64 位的积扩展到 EDX，则进位和溢出标志置位。</p> <p>指令格式：</p> <p>单操作数</p> <table><tr><td>IMUL <i>r/m8</i></td><td>IMUL <i>r/m16</i></td></tr><tr><td>IMUL <i>r/m32</i></td><td></td></tr></table> <p>双操作数</p> <table><tr><td>IMUL <i>r16, r/m16</i></td><td>IMUL <i>r16, imm8</i></td></tr><tr><td>IMUL <i>r32, r/m32</i></td><td>IMUL <i>r32, imm8</i></td></tr><tr><td>IMUL <i>r16, imm16</i></td><td>IMUL <i>r32, imm32</i></td></tr></table> <p>三操作数</p> <table><tr><td>IMUL <i>r16, r/m16, imm8</i></td><td>IMUL <i>r16, r/m16, imm16</i></td></tr><tr><td>IMUL <i>r32, r/m32, imm8</i></td><td>IMUL <i>r32, r/m32, imm32</i></td></tr></table>		O	D	I	S	Z	A	P	C	*				?	?	?	?	*	IMUL <i>r/m8</i>	IMUL <i>r/m16</i>	IMUL <i>r/m32</i>		IMUL <i>r16, r/m16</i>	IMUL <i>r16, imm8</i>	IMUL <i>r32, r/m32</i>	IMUL <i>r32, imm8</i>	IMUL <i>r16, imm16</i>	IMUL <i>r32, imm32</i>	IMUL <i>r16, r/m16, imm8</i>	IMUL <i>r16, r/m16, imm16</i>	IMUL <i>r32, r/m32, imm8</i>	IMUL <i>r32, r/m32, imm32</i>
	O	D	I	S	Z	A	P	C																									
*				?	?	?	?	*																									
IMUL <i>r/m8</i>	IMUL <i>r/m16</i>																																
IMUL <i>r/m32</i>																																	
IMUL <i>r16, r/m16</i>	IMUL <i>r16, imm8</i>																																
IMUL <i>r32, r/m32</i>	IMUL <i>r32, imm8</i>																																
IMUL <i>r16, imm16</i>	IMUL <i>r32, imm32</i>																																
IMUL <i>r16, r/m16, imm8</i>	IMUL <i>r16, r/m16, imm16</i>																																
IMUL <i>r32, r/m32, imm8</i>	IMUL <i>r32, r/m32, imm32</i>																																
IN	<p>从端口输入 (Input from Port)</p> <table><tr><td></td><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table> <p>从端口输入一个字节或字到 AL 或 AX 中。源操作数是端口地址，可以是 8 位的常量或 DX 中的一个 16 位地址。在 IA-32 处理器上，还可以从端口输入一个双字至 EAX 中。</p> <p>指令格式：</p> <table><tr><td>IN <i>accum, imm</i></td><td>IN <i>accum, DX</i></td></tr></table>		O	D	I	S	Z	A	P	C										IN <i>accum, imm</i>	IN <i>accum, DX</i>												
	O	D	I	S	Z	A	P	C																									
IN <i>accum, imm</i>	IN <i>accum, DX</i>																																
INC	<p>加 1 (Increment)</p> <table><tr><td></td><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td></td></tr></table> <p>寄存器或内存操作数加 1。</p> <p>指令格式：</p> <table><tr><td>INC <i>reg</i></td><td>INC <i>mem</i></td></tr></table>		O	D	I	S	Z	A	P	C	*				*	*	*	*		INC <i>reg</i>	INC <i>mem</i>												
	O	D	I	S	Z	A	P	C																									
*				*	*	*	*																										
INC <i>reg</i>	INC <i>mem</i>																																
INS INSB INSW INSD	<p>从端口输入一个字符串 (Input from Port to String) (80286)</p> <table><tr><td></td><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table> <p>从端口输入一个字符串至 ES:(E)DI 指向的缓冲区中。端口号在 DX 中指定，对于接收到的每个值，(E)DI 的调整方式与 LODSB 及其他类似的字符串操作指令类似。可以使用 REP 前缀。</p> <p>指令格式：</p> <table><tr><td>INS <i>dest, DX</i></td><td>REP INSB <i>dest, DX</i></td></tr><tr><td>REP INSW <i>dest, DX</i></td><td>REP INSD <i>dest, DX</i></td></tr></table>		O	D	I	S	Z	A	P	C										INS <i>dest, DX</i>	REP INSB <i>dest, DX</i>	REP INSW <i>dest, DX</i>	REP INSD <i>dest, DX</i>										
	O	D	I	S	Z	A	P	C																									
INS <i>dest, DX</i>	REP INSB <i>dest, DX</i>																																
REP INSW <i>dest, DX</i>	REP INSD <i>dest, DX</i>																																

INT	中断 (Interrupt)	<div style="display: flex; justify-content: space-around; margin-bottom: 5px;"> ODISZAPC </div> <div style="display: flex; justify-content: space-around; border: 1px solid black; padding: 5px;"> <div style="width: 20px; height: 20px;"></div> <div style="width: 20px; height: 20px;"></div> <div style="width: 20px; height: 20px; text-align: center;">0</div> <div style="width: 20px; height: 20px;"></div> <div style="width: 20px; height: 20px;"></div> <div style="width: 20px; height: 20px;"></div> <div style="width: 20px; height: 20px;"></div> <div style="width: 20px; height: 20px;"></div> </div>
	产生软件中断，导致调用操作系统的中断服务程序。在转移到中断服务程序之前，该指令清除中断标志，并将标志、CS 和 IP 压入堆栈。	
	指令格式：	
	INT imm	INT 3

INTO	溢出中断 (Interrupt on Overflow)						
	O	D	I	S	Z	A	P
			*	*			

如果设置了溢出标志, 该指令产生 4 号 CPU 中断。在 DOS 下调用 INT 4 时, 默认的服务程序不采取任何动作, 但是用户可以用自己编写的服务程序替换它。

指令格式:

INTO

IRET	中断返回 (Interrupt Return)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*	*	*	*	*	*	*	*
	O	D	I	S	Z	A	P	C									
*	*	*	*	*	*	*	*										
<p>从中断处理例程返回。从堆栈上弹出(E)IP, CS 以及标志。</p> <p>指令格式:</p> <p>IRET</p>																	

Jcondition	条件跳转 (Conditional Jump)																
	<div style="text-align: center;"><table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td><div style="border: 1px solid black; width: 20px; height: 20px;"></div></td><td><div style="border: 1px solid black; width: 20px; height: 20px;"></div></td><td><div style="border: 1px solid black; width: 20px; height: 20px;"></div></td><td><div style="border: 1px solid black; width: 20px; height: 20px;"></div></td><td><div style="border: 1px solid black; width: 20px; height: 20px;"></div></td><td><div style="border: 1px solid black; width: 20px; height: 20px;"></div></td><td><div style="border: 1px solid black; width: 20px; height: 20px;"></div></td><td><div style="border: 1px solid black; width: 20px; height: 20px;"></div></td></tr></table></div>	O	D	I	S	Z	A	P	C	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>
	O	D	I	S	Z	A	P	C									
<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>	<div style="border: 1px solid black; width: 20px; height: 20px;"></div>										
<p>如果特定的标志为真则跳转到指定的标号处。在 IA-32 处理器之前，标号距当前位置必须在 -128~+127 字节范围之内；在 IA-32 处理器上，标号距当前位置必须在 -32 768~+32 767 字节范围之内。指令助记符列表参见表 B.2。</p> <p>指令格式：</p> <p style="text-align: center;">Jcondition label</p>																	

表 B.2 条件跳转助记符^①

指令助记符	含 义	指令助记符	含 义
JA	大于则跳转	JE	相等则跳转
JNA	不大于则跳转	JNE	不等则跳转

① JA, JNA, JAE, JNAE, JB, JNB 等带“A”或者“B”的指令用于无符号数比较; JG, JNG, JL, JNL 等使用“G”或者“L”的指令用于有符号数的比较——译者注。

(续表)

指令助记符	含 义	指令助记符	含 义
JAE	大于等于则跳转	JZ	为 0 则跳转
JNAE	不大于等于则跳转	JNZ	不为 0 则跳转
JB	小于则跳转	JS	为负则跳转
JNB	不小于则跳转	JNS	不为负则跳转
JBE	小于或等于则跳转	JC	进位则跳转
JNBE	不小于等于则跳转	JNC	无进位则跳转
JG	大于则跳转	JO	溢出则跳转
JNG	不大于则跳转	JNO	未溢出则跳转
JGE	大于等于则跳转	JP	奇偶位置位则跳转
JNGE	不大于等于则跳转	JPE	奇偶位相等则跳转
JL	小于则跳转	JNP	奇偶位清除则跳转
JNL	不小于则跳转	JPO	奇偶位清除则跳转
JLE	小于或等于则跳转	JNLE	不小于等于则跳转

JCXZ JECXZ	CX 为 0 则跳转 (Jump if CX is Zero)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 1.2em; margin: 0 auto;"></div> </div>
	<p>如果 CX 寄存器等于 0 则跳转。标号距其后指令必须在-128~+127 字节范围之内。在 IA-32 处理器上, JECXZ 表示如果 ECX 等于 0 则跳转。</p> <p>指令格式:</p> <div style="display: flex; justify-content: space-between; margin-top: 10px;"> JCXZ <i>shortlabel</i> JECXZ <i>shortlabel</i> </div>

JMP	无条件跳转 (Jump Unconditionally to Label)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 1.2em; margin: 0 auto;"></div> </div>
	<p>跳转到标号处。短跳转的目的地址距当前位置应在-128~+127 字节范围之内。近跳转的目的地址应在同一代码段之内, 远跳转的目的地址可以在当前段之外。</p> <p>指令格式:</p> <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div> JMP <i>shortlabel</i> JMP <i>nearlabel</i> JMP <i>farlabel</i> </div> <div> JMP <i>reg16</i> JMP <i>mem16</i> JMP <i>mem32</i> </div> </div>

LAHF	将标志送 AH (Load AH from Flags)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 1.2em; margin: 0 auto;"></div> </div>
	<p>复制下列标志到 AH: 等于标志、零标志、辅助进位标志、奇偶标志和进位标志。</p> <p>指令格式:</p> <p style="text-align: center;">LAHF</p>

LDS LES LFS LGS LSS	<p>装入远指针 (Load Far Pointer)</p> <p style="text-align: center;">O D I S Z A P C</p> <table border="1" style="margin: auto; width: 100px; text-align: center;"><tr><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td></tr></table> <p>将双字内存操作数装入段寄存器和特定的目的寄存器中。在 IA-32 处理器之前, LDS 装入 DS, LES 装入 ES; 在 IA-32 处理器上, LFS 装入 FS, LGS 装入 GS, LSS 装入 SS。</p> <p>指令格式 (LES, LFS, LGS, LSS 相同):</p> <p style="text-align: center;">LDS <i>reg, mem</i></p>								
LEA	<p>装入有效地址 (Load Effective Address)</p> <p style="text-align: center;">O D I S Z A P C</p> <table border="1" style="margin: auto; width: 100px; text-align: center;"><tr><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td></tr></table> <p>计算并装入 16 位或 32 位的内存操作数的有效地址。类似于 MOV.EFFECTIVE, 不同点在于 LEA 指令获取的地址是在运行时进行计算的。</p> <p>指令格式:</p> <p style="text-align: center;">LEA <i>reg, mem</i></p>								
LEAVE	<p>高级过程退出 (High-Level Procedure Exit)</p> <p style="text-align: center;">O D I S Z A P C</p> <table border="1" style="margin: auto; width: 100px; text-align: center;"><tr><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td></tr></table> <p>释放过程的堆栈框架。通过将(E)SP 和(E)BP 恢复为原始值对过程开始的 ENTER 指令进行逆操作。</p> <p>指令格式:</p> <p style="text-align: center;">LEAVE</p>								
LOCK	<p>锁定系统总线 (Lock the System Bus)</p> <p style="text-align: center;">O D I S Z A P C</p> <table border="1" style="margin: auto; width: 100px; text-align: center;"><tr><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td><td> </td></tr></table> <p>在多处理器的计算机中, 其他处理器在下一条指令执行期间暂停运行。在其他处理器可能修改当前 CPU 正在访问的数据时使用该指令。</p> <p>指令格式:</p> <p style="text-align: center;">LOCK <i>instruction</i></p>								

LODS LODSB LODSW LODS	将字符串装入累加器 (Load Accumulator from String)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 20px; margin: 0 auto;"></div> </div>
	将由 DS:(E)SI 寻址的一个内存字节或字装入累加器 (AL, AX 或 EAX) 中。如果使用 LODS, 必须指定内存操作数。LODSB 将一个字节装入 AL, LODSW 将一个字装入 AX, IA-32 处理器的 LODSD 将一个双字装入 EAX。(E)SI 根据操作数大小和方向标志值自动增减。如果方向标志 DF = 1, (E)SI 增加; 如果 DF = 0, (E)SI 减少。
	指令格式: <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div>LODS <i>mem</i></div> <div>LODSB</div> </div> <div style="display: flex; justify-content: space-between; margin-top: 5px;"> <div>LODS <i>segreg:mem</i></div> <div>LODSW</div> </div> <div style="margin-top: 5px;">LODS</div>

LOOP LOOPW	循环 (Loop)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 20px; margin: 0 auto;"></div> </div>
	(E)CX 减 1, 如果 (E)CX 不等于 0 则跳转到一个短标号处。目的标号距当前位置必须在 -128 到 +127 字节范围之内。在 IA-32 上, ECX 用做默认的循环计数器。
	指令格式: <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div>LOOP <i>shortlabel</i></div> <div>LOOPW <i>shortlabel</i></div> </div>

LOOPD	循环 (Loop) (IA-32)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 20px; margin: 0 auto;"></div> </div>
	(E)CX 减 1, 如果 (E)CX 不等于 0 则跳转到一个短标号处。目的标号距当前位置必须在 -128 到 +127 字节范围之内。
	指令格式: <div style="margin-top: 10px;">LOOPD <i>shortlabel</i></div>

LOOPE LOOPZ	如果相等 (为 0) 则循环 [Loop If Equal (Zero)]
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 20px; margin: 0 auto;"></div> </div>
	(E)CX 减 1, 如果 (E)CX 大于 0 并且零标志设置则跳转到短标号处。
	指令格式: <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div>LOOPE <i>shortlabel</i></div> <div>LOOPZ <i>shortlabel</i></div> </div>

LOOPNE LOOPNZ	如果不等 (不为 0) 则循环 [Loop If Not Equal (Zero)]
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 20px; margin: 0 auto;"></div> </div>
	(E)CX 减 1, 如果 (E)CX 大于 0 并且零标志清除则跳转到短标号处。
	指令格式: <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div>LOOPNE <i>shortlabel</i></div> <div>LOOPNZ <i>shortlabel</i></div> </div>

MOV	数据传送 (Move)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 20px; margin: 0 auto;"></div> </div>
复制字节、字源操作数到目的操作数中。 指令格式： <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div> MOV <i>reg, reg</i> MOV <i>mem, reg</i> MOV <i>reg, mem</i> MOV <i>reg16, segreg</i> MOV <i>segreg, reg16</i> </div> <div> MOV <i>reg, imm</i> MOV <i>mem, imm</i> MOV <i>mem16, segreg</i> MOV <i>segreg, mem16</i> </div> </div>	

MOVS MOVSB MOVSW MOVSD	字符串传送 (Move String)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 20px; margin: 0 auto;"></div> </div>
复制 DS(E)SI 寻址的内存操作数至 ES(E)DI。MOVS 指令要求指定源和目的操作数：MOVSB 复制一个字节，MOVSW 复制一个字。在 IA-32 处理器上，MOVSD 复制一个双字。(E)SI 和 (E)DI 的值根据操作数的大小和方向标志的状态增减。如果方向标志 DF = 1, (E)SI 和 (E)DI 减少；如果 DF = 0, (E)SI 和 (E)DI 增加。 指令格式： <div style="margin-top: 10px;"> MOVSB MOVSW MOVSD MOVS <i>dest, source</i> MOVS ES:<i>dest, segreg: source</i> </div>	

MOVSX	符号扩展传送 (Move With Sign-Extend)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 20px; margin: 0 auto;"></div> </div>
将一个字节或字从源操作数复制到目的寄存器中，并将符号扩展到目的操作数的高半部分。该指令用于将一个 8 位或 16 位的操作数复制到更大的目的操作数中。 指令格式： <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div> MOVSX <i>reg32, reg16</i> MOVSX <i>reg16, reg8</i> </div> <div> MOVSX <i>reg32, mem16</i> MOVSX <i>reg16, m8</i> </div> </div>	

MOVZX	零扩展传送 (Move With Zero-Extend)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 20px; margin: 0 auto;"></div> </div>
将一个字节或字从源操作数复制到目的寄存器中，并零扩展到目的操作数的高半部分。该指令用于将一个 8 位或 16 位的操作数复制到更大的目的操作数中。 指令格式： <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div> MOVZX <i>reg32, reg16</i> MOVZX <i>reg16, reg8</i> </div> <div> MOVZX <i>reg32, mem16</i> MOVZX <i>reg16, m8</i> </div> </div>	

MUL	无符号整数乘法 (Unsigned Integer Multiply) <table border="1" style="margin: 10px auto; text-align: center;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>*</td><td></td><td></td><td>?</td><td>?</td><td>?</td><td>?</td><td>*</td></tr> </table>	O	D	I	S	Z	A	P	C	*			?	?	?	?	*
	O	D	I	S	Z	A	P	C									
*			?	?	?	?	*										
将 AL、AX 或 EAX 与源操作数相乘。如果源操作数是 8 位的, 则与 AL 相乘, 积存储在 AX 中; 如果源操作数是 16 位的, 则与 AX 相乘, 积存储在 DX:AX 中; 如果源操作数是 32 位的, 则与 EAX 相乘, 积存储在 EDX:EAX 中。 指令格式: <div style="display: flex; justify-content: space-between; margin-top: 10px;"> MUL reg MUL mem </div>																	

NEG	求补 (Negate) <table border="1" style="margin: 10px auto; text-align: center;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr> </table>	O	D	I	S	Z	A	P	C	*			*	*	*	*	*
	O	D	I	S	Z	A	P	C									
*			*	*	*	*	*										
计算目的操作数的补码并将结果存储在目的操作数中。 指令格式: <div style="display: flex; justify-content: space-between; margin-top: 10px;"> NEG reg NEG mem </div>																	

NOP	空操作 (No Operation) <table border="1" style="margin: 10px auto; text-align: center;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>	O	D	I	S	Z	A	P	C								
	O	D	I	S	Z	A	P	C									
这条指令什么也不做, 可用于计时循环中或用于后续指令的按字边界对齐。 指令格式: <div style="text-align: center; margin-top: 10px;">NOP</div>																	

NOT	求反 (Not) <table border="1" style="margin: 10px auto; text-align: center;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>	O	D	I	S	Z	A	P	C								
	O	D	I	S	Z	A	P	C									
通过将操作数的各位变反执行逻辑非操作。 指令格式: <div style="display: flex; justify-content: space-between; margin-top: 10px;"> NOT reg NOT mem </div>																	

OR	或 (Inclusive OR) <table border="1" style="margin: 10px auto; text-align: center;"> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>0</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>0</td></tr> </table>	O	D	I	S	Z	A	P	C	0			*	*	?	*	0
	O	D	I	S	Z	A	P	C									
0			*	*	?	*	0										
对目的操作数和源操作数的对应数据位执行布尔 (位) 或操作。 指令格式: <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div> OR reg, reg OR mem, reg OR reg, mem </div> <div> OR reg, imm OR mem, imm OR accum, imm </div> </div>																	

OUT	输出到端口 (Output to Port)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 15px; margin: 0 auto;"></div> </div>
<p>在 IA-32 之前, 这条指令将累加器中的一个字节或字输出到端口。端口地址如果在范围 0~FFh 之间可以是一个常量, 也可以在 DX 中存放 0~FFFFh 之间的端口地址。在 IA-32 处理器上, 可向端口输出一个双字。</p> <p>指令格式:</p> <div style="display: flex; justify-content: space-between;"> OUT imm8, accum OUT DX, accum </div>	

OUTS OUTSB OUTSW OUTSD	向端口输出字符串 (Output String to Port) (80286)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 15px; margin: 0 auto;"></div> </div>
<p>将 ES(E)DI 指向的字符串输出到端口。端口号在 DX 中指定。每输出一个值, (E)DI 按照与 LODSB 或其他字符串操作指令类似的方式进行调整。REP 前缀可以和该指令联合使用。</p> <p>指令格式:</p> <div style="display: flex; justify-content: space-between;"> OUTS dest, DX REP OUTSB dest, DX </div> <div style="display: flex; justify-content: space-between;"> REP OUTSW dest, DX REP OUTSD dest, DX </div>	

POP	出栈 (Pop from Stack)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 15px; margin: 0 auto;"></div> </div>
<p>将当前堆栈指针位置处的一个字或双字复制到目的操作数中, 并把 (E)SP 加 2 (或 4)。</p> <p>指令格式:</p> <div style="display: flex; justify-content: space-between;"> POP reg16/reg32 POP segreg </div> <div style="display: flex; justify-content: space-between;"> POP mem16/mem32 </div>	

POPA POPAD	全部出栈 (Pop All)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 15px; margin: 0 auto;"></div> </div>
<p>将堆栈顶部的 16 个字节弹出到 8 个通用寄存器中, 顺序如下: DI, SI, BP, SP, BX, DX, CX, AX。SP 的值被丢弃, 因此 SP 并不重新赋值。POPA 出栈送 16 位的寄存器中, IA-32 处理器上的 POPAD 指令出栈送 32 位的寄存器中。</p> <p>指令格式:</p> <div style="display: flex; justify-content: space-between;"> POPA POPAD </div>	

POPF POPFD	标志出栈 (Pop Flags from Stack)
	<div style="text-align: center;"> O D I S Z A P C <div style="border: 1px solid black; width: 100px; height: 15px; margin: 0 auto; position: relative;"> <div style="position: absolute; top: 0; left: 0; right: 0; bottom: 0; background-color: #f0f0f0;"></div> </div> </div>
<p>POPF 将堆栈顶部的一个字出栈送至 16 位的 FLAGS 寄存器。IA-32 上的 POPFD 将堆栈顶部的一个双字出栈送至 32 位的 EFLAGS 寄存器。</p> <p>指令格式:</p> <div style="display: flex; justify-content: space-between;"> POPF POPFD </div>	

PUSH	压栈 (Push on Stack)																							
	<div style="text-align: center;"><table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></div> <p>(E)SP 减去 2 (或 4) 并将源操作数复制到(E)SP 堆栈指针所指的堆栈位置。在 80186 以上的处理器上, 也可以将立即数压入堆栈。</p> <p>指令格式:</p> <table><tr><td>PUSH</td><td><i>reg16/reg32</i></td><td>PUSH</td><td><i>segreg</i></td></tr><tr><td>PUSH</td><td><i>mem16/mem32</i></td><td>PUSH</td><td><i>imm16/imm32</i></td></tr></table>	O	D	I	S	Z	A	P	C									PUSH	<i>reg16/reg32</i>	PUSH	<i>segreg</i>	PUSH	<i>mem16/mem32</i>	PUSH
O	D	I	S	Z	A	P	C																	
PUSH	<i>reg16/reg32</i>	PUSH	<i>segreg</i>																					
PUSH	<i>mem16/mem32</i>	PUSH	<i>imm16/imm32</i>																					

PUSHA PUSHAD	全部压栈（Push All）（80286）
	<div><div>O D I S Z A P C</div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div></div>
	<p>在堆栈中按顺序压入下列 16 位的寄存器：AX，CX，DX，BX，SP，BP，SI 和 DI。IA-32 的 PUSHAD 指令压入 EAX，ECX，EDX，EBX，ESP，EBP，ESI 和 EDI。</p> <p>指令格式：</p> <div><div>PUSHA</div><div>PUSHAD</div></div>

PUSHF PUSHFD	标志压栈 (Push Flags)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C								
	O	D	I	S	Z	A	P	C									
<p>PUSHF 在堆栈上压入 16 位的 FLAGS 寄存器。PUSHFD (IA-32) 在堆栈上压入 32 位的 EFLAGS 寄存器。</p> <p>指令格式:</p> <table><tr><td>PUSHF</td><td>PUSHFD</td></tr></table>	PUSHF	PUSHFD															
PUSHF	PUSHFD																

PUSHW PUSHD	压栈 (Push on Stack)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C							
O	D	I	S	Z	A	P	C									
PUSHW 在堆栈上压入一个 16 位的字。在 IA-32 上, PUSHD 在堆栈上压入一个 32 位双字。 指令格式:																
PUSH <i>reg16/reg32</i>	PUSH <i>segreg</i>															
PUSH <i>mem16/mem32</i>	PUSH <i>imm16/imm32</i>															

RCL	带进位循环左移 (Rotate Carry Left)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td></td><td></td><td></td><td></td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*							*
	O	D	I	S	Z	A	P	C									
*							*										
目的操作数循环左移，源操作数决定移位的数量。进位的值复制到最低位，最高位送进位标志中。使用 8086/8088 处理器时，imm8 操作数必须是 1。 指令格式： <table><tr><td>RCL</td><td>reg, imm8</td><td>RCL</td><td>mem, imm8</td></tr><tr><td>RCL</td><td>reg, CL</td><td>RCL</td><td>mem, CL</td></tr></table>	RCL	reg, imm8	RCL	mem, imm8	RCL	reg, CL	RCL	mem, CL									
RCL	reg, imm8	RCL	mem, imm8														
RCL	reg, CL	RCL	mem, CL														

RCR	<p>带进位循环右移 (Rotate Carry Right)</p> <table border="1"><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td></td><td></td><td></td><td></td><td>*</td></tr></table> <p>目的操作数循环右移，源操作数决定移位的数量。进位的值复制到最高位，最低位送进位标志中。使用 8086/8088 处理器时，imm8 操作数必须是 1。</p> <p>指令格式：</p> <table><tr><td>RCR</td><td>reg, imm8</td><td>RCR</td><td>mem, imm8</td></tr><tr><td>RCR</td><td>reg, CL</td><td>RCR</td><td>mem, CL</td></tr></table>	O	D	I	S	Z	A	P	C	*							*	RCR	reg, imm8	RCR	mem, imm8	RCR	reg, CL	RCR	mem, CL
O	D	I	S	Z	A	P	C																		
*							*																		
RCR	reg, imm8	RCR	mem, imm8																						
RCR	reg, CL	RCR	mem, CL																						

REP	<p>重复字符串操作 (Repeat String)</p> <table border="1"><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table> <p>使用(E)CX 作为计数器重复字符串操作指令。每次指令重复的时候(E)CX 减 1，直到(E)CX 等于 0。</p> <p>指令格式 (以 MOVS 为例)：</p> <p>REP MOVS dest, source</p>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										

REPcondition	<p>有条件重复字符串操作 (Repeat String Conditionally)</p> <table border="1"><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table> <p>在标志条件为真时重复字符串操作直到(E)CX = 0。REPZ (REPE) 当零标志设置时重复，REPNZ 和 REPNE 当零标志清除时重复。只有 SCAS 和 CMPS 才能和 REPcondition 联合使用，因为它们是惟一的修改零标志的字符串操作指令。</p> <p>指令格式 (以 SCAS 为例)：</p> <table><tr><td>REPZ</td><td>SCAS</td><td>dest</td><td>REPNE</td><td>SCAS</td><td>dest</td></tr><tr><td>REPZ</td><td>SCASB</td><td></td><td>REPNE</td><td>SCASB</td><td></td></tr><tr><td>REPE</td><td>SCASW</td><td></td><td>REPNZ</td><td>SCASW</td><td></td></tr></table>	O	D	I	S	Z	A	P	C									REPZ	SCAS	dest	REPNE	SCAS	dest	REPZ	SCASB		REPNE	SCASB		REPE	SCASW		REPNZ	SCASW	
O	D	I	S	Z	A	P	C																												
REPZ	SCAS	dest	REPNE	SCAS	dest																														
REPZ	SCASB		REPNE	SCASB																															
REPE	SCASW		REPNZ	SCASW																															

RET RETN RETF	<p>过程返回 (Return from Procedure)</p> <table border="1"><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table> <p>从堆栈上弹出返回地址。RETN (近返回) 只从堆栈顶部弹出(E)IP。在实地址模式下，RETF (远返回) 首先弹出(E)IP，然后弹出 CS。根据 PROC 伪指令指定的或暗含的属性的不同，RET 可以是近的或远的。可选的 8 位操作数通知 CPU 在弹出返回地址后给(E)SP 加上一个值。</p> <p>指令格式：</p> <table><tr><td>RET</td><td></td><td>RET</td><td>imm8</td></tr><tr><td>RETN</td><td></td><td>RETN</td><td>imm8</td></tr><tr><td>RETF</td><td></td><td>RETF</td><td>imm8</td></tr></table>	O	D	I	S	Z	A	P	C									RET		RET	imm8	RETN		RETN	imm8	RETF		RETF	imm8
O	D	I	S	Z	A	P	C																						
RET		RET	imm8																										
RETN		RETN	imm8																										
RETF		RETF	imm8																										

ROL	<p>循环左移 (Rotate Left)</p> <table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td></td><td></td><td></td><td></td><td>*</td></tr></table> <p>目的操作数循环左移, 源操作数决定移位的数目。最高位复制到进位标志中并同时送至最低位中。使用 8086/8088 处理器时, imm8 操作数必须是 1。</p> <p>指令格式:</p> <table><tr><td>ROL</td><td>reg, imm8</td><td>ROL</td><td>mem, imm8</td></tr><tr><td>ROL</td><td>reg, CL</td><td>ROL</td><td>mem, CL</td></tr></table>	O	D	I	S	Z	A	P	C	*							*	ROL	reg, imm8	ROL	mem, imm8	ROL	reg, CL	ROL	mem, CL
O	D	I	S	Z	A	P	C																		
*							*																		
ROL	reg, imm8	ROL	mem, imm8																						
ROL	reg, CL	ROL	mem, CL																						
ROR	<p>循环右移 (Rotate Right)</p> <table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td></td><td></td><td></td><td></td><td>*</td></tr></table> <p>目的操作数循环右移, 源操作数决定移位的数目。最低位复制到进位标志中并同时送至最高位。使用 8086/8088 处理器时, imm8 操作数必须是 1。</p> <p>指令格式:</p> <table><tr><td>ROR</td><td>reg, imm8</td><td>ROR</td><td>mem, imm8</td></tr><tr><td>ROR</td><td>reg, CL</td><td>ROR</td><td>mem, CL</td></tr></table>	O	D	I	S	Z	A	P	C	*							*	ROR	reg, imm8	ROR	mem, imm8	ROR	reg, CL	ROR	mem, CL
O	D	I	S	Z	A	P	C																		
*							*																		
ROR	reg, imm8	ROR	mem, imm8																						
ROR	reg, CL	ROR	mem, CL																						
SAHF	<p>AH 送标志寄存器 (Store AH Into Flags)</p> <table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table> <p>复制 AH 到标志寄存器的位 0 到 7 中。</p> <p>指令格式:</p> <p>SAHF</p>	O	D	I	S	Z	A	P	C				*	*	*	*	*								
O	D	I	S	Z	A	P	C																		
			*	*	*	*	*																		
SAL	<p>算术左移 (Shift Arithmetic Left)</p> <table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>*</td></tr></table> <p>目的操作数中的每一位左移, 源操作数决定移位数目。最高位复制到进位标志中, 最低位以 0 填充。使用 8086/8088 处理器时, imm8 操作数必须是 1。</p> <p>指令格式:</p> <table><tr><td>SAL</td><td>reg, imm8</td><td>SAL</td><td>mem, imm8</td></tr><tr><td>SAL</td><td>reg, CL</td><td>SAL</td><td>mem, CL</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	?	*	*	SAL	reg, imm8	SAL	mem, imm8	SAL	reg, CL	SAL	mem, CL
O	D	I	S	Z	A	P	C																		
*			*	*	?	*	*																		
SAL	reg, imm8	SAL	mem, imm8																						
SAL	reg, CL	SAL	mem, CL																						
SAR	<p>算术右移 (Shift Arithmetic Right)</p> <table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>*</td></tr></table> <p>目的操作数中的每一位右移, 源操作数决定移位数目。最低位复制到进位标志中, 最高位保持原值。SAR 指令通常用于有符号数操作, 因为它保留了符号位的值。使用 8086/8088 处理器时, imm8 操作数必须是 1。</p> <p>指令格式:</p> <table><tr><td>SAR</td><td>reg, imm8</td><td>SAR</td><td>mem, imm8</td></tr><tr><td>SAR</td><td>reg, CL</td><td>SAR</td><td>mem, CL</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	?	*	*	SAR	reg, imm8	SAR	mem, imm8	SAR	reg, CL	SAR	mem, CL
O	D	I	S	Z	A	P	C																		
*			*	*	?	*	*																		
SAR	reg, imm8	SAR	mem, imm8																						
SAR	reg, CL	SAR	mem, CL																						

SBB	带进位减 (Subtract With Borrow)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	*	*	*
	O	D	I	S	Z	A	P	C									
*			*	*	*	*	*										
<p>从源操作数中减去目的操作数，然后再减去进位标志值。</p> <p>指令格式：</p> <table><tr><td>SBB</td><td>reg, reg</td><td>SBB</td><td>reg, imm</td></tr><tr><td>SBB</td><td>mem, reg</td><td>SBB</td><td>mem, imm</td></tr><tr><td>SBB</td><td>reg, mem</td><td></td><td></td></tr></table>	SBB	reg, reg	SBB	reg, imm	SBB	mem, reg	SBB	mem, imm	SBB	reg, mem							
SBB	reg, reg	SBB	reg, imm														
SBB	mem, reg	SBB	mem, imm														
SBB	reg, mem																

SCAS	扫描字符串 (Scan String)
SCASB	
SCASW	
SCASD	

O	D	I	S	Z	A	P	C
*			*	*	*	*	*

扫描 ES:(E)DI 指向的内存字符串查找与累加器匹配的值。SCAS 要求指定操作数。SCASB 扫描查找与 AL 匹配的 8 位值；SCASW 扫描与 AX 匹配的 16 位值；SCASD 扫描与 EAX 匹配的 32 位值。(E)DI 依据操作数的大小和方向标志的值自动增减。如果 DF = 1, (E)DI 减少；如果 DF = 0, (E)DI 增加。

指令格式：

<p>SCASB</p> <p>SCASD</p> <p>SCAS <i>dest</i></p>	<p>SCASW</p> <p>SCAS ES:<i>dest</i></p>
---	---

SETcondition	条件设置 (Set Conditionally)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td><input type="checkbox"/></td><td><input type="checkbox"/></td><td><input type="checkbox"/></td><td><input type="checkbox"/></td><td><input type="checkbox"/></td><td><input type="checkbox"/></td><td><input type="checkbox"/></td><td><input type="checkbox"/></td></tr></table>	O	D	I	S	Z	A	P	C	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
O	D	I	S	Z	A	P	C									
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>									
	<p>如果给定的条件为真，目的操作数指定的字节被赋予值 1；如果标志条件为假，目的被赋予值 0。条件的可能值列在本附录前面的表 B.2 中。</p> <p>指令格式：</p> <div><div>SETcond reg8</div><div>SETcond mem8</div></div>															

SHL	<p>逻辑左移(Shift Left)</p> <div style="text-align: center;"> <table> <tr> <td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr> <tr> <td>*</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>*</td></tr> </table> </div> <p>将目的操作数的每位左移，使用源操作数决定要移位的数目。最高位复制到进位标志中，最低位以 0 填充（与 SAL 相同）。在使用 8086/8088 处理器时，imm8 操作数必须是 1。</p> <p>指令格式：</p> <table style="width: 100%;"> <tr> <td style="width: 50%;">SHL <i>reg</i>, imm8</td><td style="width: 50%;">SHL <i>mem</i>, imm8</td></tr> <tr> <td>SHL <i>reg</i>, CL</td><td>SHL <i>mem</i>, CL</td></tr> </table>	O	D	I	S	Z	A	P	C	*			*	*	?	*	*	SHL <i>reg</i> , imm8	SHL <i>mem</i> , imm8	SHL <i>reg</i> , CL	SHL <i>mem</i> , CL
O	D	I	S	Z	A	P	C														
*			*	*	?	*	*														
SHL <i>reg</i> , imm8	SHL <i>mem</i> , imm8																				
SHL <i>reg</i> , CL	SHL <i>mem</i> , CL																				

SHLD	双精度左移 (Double-Precision Shift Left) (IA-32)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	?	*
O	D	I	S	Z	A	P	C									
*			*	*	?	*	*									
<p>将第二个操作数的若干位移到第一个操作数中, 第三个操作数表示要移位的数目。第一个操作数移出的位由第二个操作数的高位填充。第二个操作数必须是寄存器, 第三个操作数可以是立即数或 CL 寄存器。</p> <p>指令格式:</p> <table><tr><td>SHLD <i>reg16, reg16, imm8</i></td><td>SHLD <i>mem16, reg16, imm8</i></td></tr><tr><td>SHLD <i>reg32, reg32, imm8</i></td><td>SHLD <i>mem32, reg32, imm8</i></td></tr><tr><td>SHLD <i>reg16, reg16, CL</i></td><td>SHLD <i>mem16, reg16, CL</i></td></tr><tr><td>SHLD <i>reg32, reg32, CL</i></td><td>SHLD <i>mem32, reg32, CL</i></td></tr></table>		SHLD <i>reg16, reg16, imm8</i>	SHLD <i>mem16, reg16, imm8</i>	SHLD <i>reg32, reg32, imm8</i>	SHLD <i>mem32, reg32, imm8</i>	SHLD <i>reg16, reg16, CL</i>	SHLD <i>mem16, reg16, CL</i>	SHLD <i>reg32, reg32, CL</i>	SHLD <i>mem32, reg32, CL</i>							
SHLD <i>reg16, reg16, imm8</i>	SHLD <i>mem16, reg16, imm8</i>															
SHLD <i>reg32, reg32, imm8</i>	SHLD <i>mem32, reg32, imm8</i>															
SHLD <i>reg16, reg16, CL</i>	SHLD <i>mem16, reg16, CL</i>															
SHLD <i>reg32, reg32, CL</i>	SHLD <i>mem32, reg32, CL</i>															

SHR	右移(Shift Right)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	?	*	*
	O	D	I	S	Z	A	P	C									
*			*	*	?	*	*										
<p>目的操作数中的每一位右移, 使用源操作数决定移位的数目。最高位以 0 填充, 最低位复制到进位标志中。在使用 8086/8088 处理器时, imm8 必须是 1。</p> <p>指令格式:</p> <table><tr><td>SHR <i>reg, imm8</i></td><td>SHR <i>mem, imm8</i></td></tr><tr><td>SHR <i>reg, CL</i></td><td>SHR <i>mem, CL</i></td></tr></table>	SHR <i>reg, imm8</i>	SHR <i>mem, imm8</i>	SHR <i>reg, CL</i>	SHR <i>mem, CL</i>													
SHR <i>reg, imm8</i>	SHR <i>mem, imm8</i>																
SHR <i>reg, CL</i>	SHR <i>mem, CL</i>																

SHRD	双精度右移 (Double-Precision Shift Right) (IA-32)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	?	*
O	D	I	S	Z	A	P	C									
*			*	*	?	*	*									
<p>将第二个操作数的若干位移到第一个操作数中, 第三个操作数表示要移位的数目。第一个操作数移出的位由第二个操作数的低位填充。第二个操作数必须是寄存器, 第三个操作数可以是立即数或 CL 寄存器。</p> <p>指令格式:</p> <table><tr><td>SHRD <i>reg16, reg16, imm8</i></td><td>SHRD <i>mem16, reg16, imm8</i></td></tr><tr><td>SHRD <i>reg32, reg32, imm8</i></td><td>SHRD <i>mem32, reg32, imm8</i></td></tr><tr><td>SHRD <i>reg16, reg16, CL</i></td><td>SHRD <i>mem16, reg16, CL</i></td></tr><tr><td>SHRD <i>reg32, reg32, CL</i></td><td>SHRD <i>mem32, reg32, CL</i></td></tr></table>		SHRD <i>reg16, reg16, imm8</i>	SHRD <i>mem16, reg16, imm8</i>	SHRD <i>reg32, reg32, imm8</i>	SHRD <i>mem32, reg32, imm8</i>	SHRD <i>reg16, reg16, CL</i>	SHRD <i>mem16, reg16, CL</i>	SHRD <i>reg32, reg32, CL</i>	SHRD <i>mem32, reg32, CL</i>							
SHRD <i>reg16, reg16, imm8</i>	SHRD <i>mem16, reg16, imm8</i>															
SHRD <i>reg32, reg32, imm8</i>	SHRD <i>mem32, reg32, imm8</i>															
SHRD <i>reg16, reg16, CL</i>	SHRD <i>mem16, reg16, CL</i>															
SHRD <i>reg32, reg32, CL</i>	SHRD <i>mem32, reg32, CL</i>															

STC	设置进位标志 (Set Carry Flag)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td></tr></table>	O	D	I	S	Z	A	P	C							
O	D	I	S	Z	A	P	C									
							1									
	设置进位标志。 指令格式： STC															

STD	<p>设置方向标志 (Set Direction Flag)</p> <div style="text-align: center; margin: 10px 0;"> <table style="margin: auto; border-collapse: collapse;"> <tr> <td style="padding: 0 10px;">O</td> <td style="padding: 0 10px;">D</td> <td style="padding: 0 10px;">I</td> <td style="padding: 0 10px;">S</td> <td style="padding: 0 10px;">Z</td> <td style="padding: 0 10px;">A</td> <td style="padding: 0 10px;">P</td> <td style="padding: 0 10px;">C</td> </tr> <tr> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px; text-align: center;">1</td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> </tr> </table> </div> <p>设置方向标志, 导致字符串操作指令自动减少(E)SI 和/或(E)DI 的值, 这样, 字符串处理就能从高地址向低地址进行。</p> <p>指令格式:</p> <p style="text-align: center; margin-top: 10px;">STD</p>	O	D	I	S	Z	A	P	C		1						
O	D	I	S	Z	A	P	C										
	1																

STI	<p>设置中断标志 (Set Interrupt Flag)</p> <div style="text-align: center; margin: 10px 0;"> <table style="margin: auto; border-collapse: collapse;"> <tr> <td style="padding: 0 10px;">O</td> <td style="padding: 0 10px;">D</td> <td style="padding: 0 10px;">I</td> <td style="padding: 0 10px;">S</td> <td style="padding: 0 10px;">Z</td> <td style="padding: 0 10px;">A</td> <td style="padding: 0 10px;">P</td> <td style="padding: 0 10px;">C</td> </tr> <tr> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px; text-align: center;">1</td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> </tr> </table> </div> <p>设置中断标志, 以允许可屏蔽硬件中断。中断发生时自动禁止中断, 因此中断处理程序应使用 STI 立即重新允许中断。</p> <p>指令格式:</p> <p style="text-align: center; margin-top: 10px;">STI</p>	O	D	I	S	Z	A	P	C			1					
O	D	I	S	Z	A	P	C										
		1															

STOS STOSB STOSW STOSD	<p>存储字符串数据 (Store String Data)</p> <div style="text-align: center; margin: 10px 0;"> <table style="margin: auto; border-collapse: collapse;"> <tr> <td style="padding: 0 10px;">O</td> <td style="padding: 0 10px;">D</td> <td style="padding: 0 10px;">I</td> <td style="padding: 0 10px;">S</td> <td style="padding: 0 10px;">Z</td> <td style="padding: 0 10px;">A</td> <td style="padding: 0 10px;">P</td> <td style="padding: 0 10px;">C</td> </tr> <tr> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> </tr> </table> </div> <p>将累加器内容存储到由 ES:(E)DI 寻址的内存地址。如果使用 STOS, 必须指定目的操作数。STOSB 复制 AL 到内存中, STOSW 复制 AX 到内存中, STOSD 复制 EAX 到内存中。(E)DI 根据操作数的尺寸和方向标志的状态值增减。如果 DF = 1, (E)DI 增加; 如果 DF = 0, (E)DI 减少。</p> <p>指令格式:</p> <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div style="width: 45%;"> <p style="margin: 5px 0;">STOSB</p> <p style="margin: 5px 0;">STOSD</p> <p style="margin: 5px 0;">STOS mem</p> </div> <div style="width: 45%;"> <p style="margin: 5px 0;">STOSW</p> <p style="margin: 5px 0;">STOS ES:mem</p> </div> </div>	O	D	I	S	Z	A	P	C								
O	D	I	S	Z	A	P	C										

SUB	<p>减法 (Subtract)</p> <div style="text-align: center; margin: 10px 0;"> <table style="margin: auto; border-collapse: collapse;"> <tr> <td style="padding: 0 10px;">O</td> <td style="padding: 0 10px;">D</td> <td style="padding: 0 10px;">I</td> <td style="padding: 0 10px;">S</td> <td style="padding: 0 10px;">Z</td> <td style="padding: 0 10px;">A</td> <td style="padding: 0 10px;">P</td> <td style="padding: 0 10px;">C</td> </tr> <tr> <td style="border: 1px solid black; width: 20px; height: 20px; text-align: center;">*</td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px;"></td> <td style="border: 1px solid black; width: 20px; height: 20px; text-align: center;">*</td> <td style="border: 1px solid black; width: 20px; height: 20px; text-align: center;">*</td> <td style="border: 1px solid black; width: 20px; height: 20px; text-align: center;">*</td> <td style="border: 1px solid black; width: 20px; height: 20px; text-align: center;">*</td> <td style="border: 1px solid black; width: 20px; height: 20px; text-align: center;">*</td> </tr> </table> </div> <p>从目的操作数中减去源操作数。</p> <p>指令格式:</p> <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div style="width: 45%;"> <p style="margin: 5px 0;">SUB reg, reg</p> <p style="margin: 5px 0;">SUB mem, reg</p> <p style="margin: 5px 0;">SUB reg, mem</p> </div> <div style="width: 45%;"> <p style="margin: 5px 0;">SUB reg, imm</p> <p style="margin: 5px 0;">SUB mem, imm</p> <p style="margin: 5px 0;">SUB accum, imm</p> </div> </div>	O	D	I	S	Z	A	P	C	*			*	*	*	*	*
O	D	I	S	Z	A	P	C										
*			*	*	*	*	*										

TEST	测试 (Test)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>0</td><td></td><td></td><td>*</td><td>*</td><td>?</td><td>*</td><td>0</td></tr></table>	O	D	I	S	Z	A	P	C	0			*	*	?	*	0
	O	D	I	S	Z	A	P	C									
0			*	*	?	*	0										
<p>测试目的操作数的单个位。指令执行逻辑与操作，影响标志值但不改变目的操作数的内容。</p> <p>指令格式：</p> <table><tr><td>TEST</td><td>reg, reg</td><td>TEST</td><td>reg, imm</td></tr><tr><td>TEST</td><td>mem, reg</td><td>TEST</td><td>mem, imm</td></tr><tr><td>TEST</td><td>reg, mem</td><td>TEST</td><td>accum, imm</td></tr></table>	TEST	reg, reg	TEST	reg, imm	TEST	mem, reg	TEST	mem, imm	TEST	reg, mem	TEST	accum, imm					
TEST	reg, reg	TEST	reg, imm														
TEST	mem, reg	TEST	mem, imm														
TEST	reg, mem	TEST	accum, imm														

WAIT	等待协处理器 (Wait for Coprocessor)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C								
	O	D	I	S	Z	A	P	C									
挂起 CPU，直到协处理器完成当前的指令。 指令格式： WAIT																	

XADD	加交换 (Exchange and Add) (Intel 486)																
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td>*</td><td></td><td></td><td>*</td><td>*</td><td>*</td><td>*</td><td>*</td></tr></table>	O	D	I	S	Z	A	P	C	*			*	*	*	*	*
	O	D	I	S	Z	A	P	C									
*			*	*	*	*	*										
<p>源操作数与目的操作数相加。同时，原始的目的操作数送至源操作数。</p> <p>指令格式：</p> <table><tr><td>XADD</td><td>reg, reg</td><td>XADD</td><td>mem, reg</td></tr></table>	XADD	reg, reg	XADD	mem, reg													
XADD	reg, reg	XADD	mem, reg														

XCHG	交换 (Exchange)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C							
O	D	I	S	Z	A	P	C									
交换源操作数和目的操作数的内容。																
指令格式:																
XCH <i>reg, reg</i>	XCH <i>mem, reg</i>															
XCH <i>reg, mem</i>																

XLAT XLATB	字节转换 (Translate Byte)															
	<table><tr><td>O</td><td>D</td><td>I</td><td>S</td><td>Z</td><td>A</td><td>P</td><td>C</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	O	D	I	S	Z	A	P	C							
O	D	I	S	Z	A	P	C									
<p>使用 AL 中的值作为由 DS:BX 指向的一张表的索引，该索引指向的字节送至 AL。可以使用操作数指定一个段超越前缀。XLATB 可以用 XLAT 代替。</p> <p>指令格式：</p> <table><tr><td>XLAT</td><td></td><td>XLAT</td><td>segreg:mem</td></tr><tr><td>XLAT</td><td>mem</td><td>XLATB</td><td></td></tr></table>		XLAT		XLAT	segreg:mem	XLAT	mem	XLATB								
XLAT		XLAT	segreg:mem													
XLAT	mem	XLATB														

XOR	异或 (Exclusive OR)							
	O	D	I	S	Z	A	P	C
	0			*	*	?	*	0
	源操作数的每位同目的操作数的对应位进行异或操作。只有当原始操作数的数据位与目的操作数的对应位不同时结果才为 1。							
	指令格式:							
	XOR <i>reg, reg</i>				XOR <i>reg, imm</i>			
	XOR <i>mem, reg</i>				XOR <i>mem, imm</i>			
	XOR <i>reg, mem</i>				XOR <i>accum, imm</i>			

B.3 浮点指令

表 B.3 列出了所有的 IA-32 指令并简要说明了其用途以及操作数格式。表中的指令顺序不是完全按照严格的字母顺序排序的, 在有些时候优先按照指令的功能分组。例如, FIADD 指令紧跟在 FADD 和 FADDP 指令之后, 这是由于 FIADD 除了要多进行一步整数转换外, 执行的其他操作和它们是相同的。

更多关于 IA-32 浮点指令的信息, 请参考 *Intel Architecture Software Developer's Manual*, 卷 II。表中的堆栈 (stack) 是指 FPU 寄存器堆栈。(在描述浮点指令格式及操作数时使用的部分符号请参见表 B.1。)

表 B.3 IA-32 浮点指令

指 令	描 述
F2XMI	计算 $2^x - 1$, 无操作数
FABS	绝对值。清除 ST(0) 的符号位, 无操作数
FADD	浮点数加法。目的操作数和源操作数相加, 和保存在目的操作数中。格式: <div> <div>FADD</div> <div>Add ST(0) to ST(1), and pop stack</div> <div>FADD <i>m32fp</i></div> <div>Add <i>m32fp</i> to ST(0)</div> <div>FADD <i>m64fp</i></div> <div>Add <i>m64fp</i> to ST(0)</div> <div>FADD ST(0), ST(i)</div> <div>Add ST(i) to ST(0)</div> <div>FADD ST(i), ST(0)</div> <div>Add ST(0) to ST(i)</div> </div>
FADDP	浮点数相加并出栈。执行的操作与 FADD 相同, 最后 ST(0) 出栈。格式: <div> <div>FADDP ST(i), ST(0)</div> <div>Add ST(0) to ST(i)</div> </div>
FIADD	把整数转换成浮点数并相加。目的操作数和源操作数相加, 和保存在目的操作数中。格式: <div> <div>FIADD <i>m32int</i></div> <div>Add <i>m32int</i> to ST(0)</div> <div>FIADD <i>m16int</i></div> <div>Add <i>m16int</i> to ST(0)</div> </div>
FBLD	加载 BCD 数。把 BCD 格式的源操作数转换成扩展精度浮点数格式并压栈。格式: <div> <div>FBLD <i>m80bcd</i></div> <div>Push <i>m80bcd</i> onto register stack</div> </div>
FBSTP	存储 BCD 整数并出栈。把 ST(0) 中的值转换成 18 位的压缩 BCD 整数并存储在目的操作数中, 最后执行出栈操作。格式: <div> <div>FBSTP <i>m80bcd</i></div> <div>Store ST(0) into <i>m80bcd</i>, and pop stack</div> </div>
FCHS	改变符号。把 ST(0) 中值的符号变反, 无操作数

(续表)

指 令	描 述
FCLEX	清除异常。清除浮点异常标志 (PE, UE, OE, ZE, DE, IE)、标志 ES、堆栈错误标志 (SF) 以及 FPU 状态字中的忙 (B) 标志。无操作数。FNCLEX 执行同样的操作, 但不检查未决的未屏蔽异常
FCMOV _{cc}	浮点条件传送。测试 EFLAGS 状态标志, 如果条件满足, 则把源 (第二个) 操作数送目的 (第一个) 操作数。格式: <div style="margin-left: 40px;"> FCMOVB ST(0),ST(i) Move if below FCMOVE ST(0),ST(i) Move if equal FCMOVBE ST(0),ST(i) Move if below or equal FCMOVU ST(0),ST(i) Move if unordered FCMOVNB ST(0),ST(i) Move if not below FCMOVNE ST(0),ST(i) Move if not equal FCMOVNBE ST(0),ST(i) Move if not below or equal FCMOVNU ST(0),ST(i) Move if not unordered </div>
FCOM	比较浮点值。比较 ST(0)和源操作数并根据结果相应设置 FPU 状态字中的 C0, C2, C3 标志。格式: <div style="margin-left: 40px;"> FCOM <i>m32fp</i> Compare ST(0) to <i>m32fp</i> FCOM <i>m64fp</i> Compare ST(0) to <i>m64fp</i> FCOM ST(i) Compare ST(0) to ST(i) FCOM Compare ST(0) to ST(1) </div> FCOMP 执行同样的操作, 区别在于前者最后要执行出栈操作; FCOMPP 执行同样的操作, 区别在于要执行两次出栈操作; FUCOM, FUCOMP, FUCOMPP 除了检查无序数之外, 执行的操作分别与 FCOM, FCOMP, FCOMPP 相同
FCOMI	比较浮点值并设置 EFLAGS 标志。比较 ST(0)和 ST(i)并根据结果设置 EFLAGS 中的状态标志 (ZF, PF, CF)。格式: <div style="margin-left: 40px;">FCOMI ST(0),ST(i) Compare ST(0) to ST(i)</div> FCOMIP 执行的操作与 FCOMI 基本相同, 区别在于前者最后要执行出栈操作; FUCOMI 和 FUCOMIP 执行的操作与 FCOMI 和 FCOMIP 基本相同, 区别在于前二者要检查无序数
FCOS	计算余弦值。计算 ST(0)中的余弦值并把结果存储在 ST(0)中, 输入必须是弧度值, 无操作数
FDECSTP	栈顶指针减 1。FPU 状态字的 TOP 域减 1, 无操作数
FDIV	浮点除法并出栈。目的操作数除以源操作数并把结果存储在目的操作数位置。格式: <div style="margin-left: 40px;"> FDIV ST(1) = ST(1) / ST(0), and pop stack FDIV <i>m32fp</i> ST(0) = ST(0) / <i>m32fp</i> FDIV <i>m64fp</i> ST(0) = ST(0) / <i>m64fp</i> FDIV ST(0),ST(i) ST(0) = ST(0) / ST(i) FDIV ST(i),ST(0) ST(i) = ST(i) / ST(0) </div>
FDIVP	浮点除法并出栈。同 FDIV, 要执行出栈操作。格式: <div style="margin-left: 40px;">FDIVP ST(i),ST(0) ST(i) = ST(i) / ST(0), and pop stack</div>
FIDIV	整数转换成浮点数并执行除法操作。在转换后, 执行的操作与 FDIV 相同。格式: <div style="margin-left: 40px;"> FIDIV <i>m32int</i> ST(0) = ST(0) / <i>m32int</i> FIDIV <i>m16int</i> ST(0) = ST(0) / <i>m16int</i> </div>

(续表)

指 令	描 述
FDIVR	<p>逆向除法。源操作数除以目的操作数并把结果存储在目的操作数中。格式：</p> <p>FDIVR $ST(0) = ST(0) / ST(1)$, and pop stack</p> <p>FDIVR <i>m32fp</i> $ST(0) = m32fp / ST(0)$</p> <p>FDIVR <i>m64fp</i> $ST(0) = m64fp / ST(0)$</p> <p>FDIVR $ST(0), ST(i)$ $ST(0) = ST(i) / ST(0)$</p> <p>FDIVR $ST(i), ST(0)$ $ST(i) = ST(0) / ST(i)$</p>
FDIVRP	<p>逆向除法并出栈。执行与 FDIVR 同样的操作，然后再出栈。格式：</p> <p>FDIVRP $ST(i), ST(0)$ $ST(i) = ST(0) / ST(i)$, and pop stack</p>
FIDIVR	<p>把整数转换成浮点数并执行反向除法。转换之后执行与 FDIVR 同样的操作。格式：</p> <p>FIDIVR <i>m32int</i> $ST(0) = m32int / ST(0)$</p> <p>FIDIVR <i>m16int</i> $ST(0) = m16int / ST(0)$</p>
FFREE	<p>释放浮点寄存器。使用标记字把寄存器设为空。格式：</p> <p>FFREE $ST(i)$ $ST(i) = \text{empty}$</p>
FICOM	<p>比较整数。比较 $ST(0)$ 和源整数操作数的值，根据结果相应设置 C0, C2, C3 标志。在比较之前，整数源操作数首先转换成浮点数。格式：</p> <p>FICOM <i>m32int</i> Compare $ST(0)$ to <i>m32int</i></p> <p>FICOM <i>m16int</i> Compare $ST(0)$ to <i>m16int</i></p>
FILD	<p>把整数转换成浮点数并加载至寄存器栈。格式：</p> <p>FILD <i>m16int</i> Push <i>m16int</i> onto register stack</p> <p>FILD <i>m32int</i> Push <i>m32int</i> onto register stack</p> <p>FILD <i>m64int</i> Push <i>m64int</i> onto register stack</p>
FINCSTP	栈顶指针加 1。FPU 状态字的 TOP 域加 1。无操作数
FINIT	<p>初始化浮点单元。把控制寄存器、状态寄存器、标记寄存器、指令指针寄存器、数据指针寄存器设为默认值。控制字设为 037FH (近似到最近，屏蔽所有异常，64 位精度)；状态字清零 (无异常标志位置位，TOP = 0)；寄存器栈中的数据寄存器内容不会改变，但标记为空。无操作数。FNINIT 执行的操作相同，区别在于不检查未决的未屏蔽异常</p>
FIST	<p>存储整数至内存。把 $ST(0)$ 存储在有符号整数内存操作数中，近似方法基于 FPU 控制字中的 RC 域。格式：</p> <p>FIST <i>m16int</i> Store $ST(0)$ in <i>m16int</i></p> <p>FIST <i>m32int</i> Store $ST(0)$ in <i>m32int</i></p> <p>FISTP 执行的操作与 FIST 基本相同，不过最后还要执行一次出栈操作。它有下列的另外一种格式：</p> <p>FISTP <i>m64int</i> Store $ST(0)$ in <i>m64int</i>, and pop stack</p>
FISTTP	<p>存储整数并剪裁。执行的操作与 FIST 相同，区别在于对整数进行剪裁并出栈。格式：</p> <p>FISTTP <i>m16int</i> Store $ST(0)$ in <i>m16int</i>, and pop stack</p> <p>FISTTP <i>m32int</i> Store $ST(0)$ in <i>m32int</i>, and pop stack</p> <p>FISTTP <i>m64int</i> Store $ST(0)$ in <i>m64int</i>, and pop stack</p>
FLD	<p>加载浮点值至寄存器栈。格式：</p> <p>FLD <i>m32fp</i> Push <i>m32fp</i> onto register stack</p> <p>FLD <i>m64fp</i> Push <i>m64fp</i> onto register stack</p> <p>FLD <i>m80fp</i> Push <i>m80fp</i> onto register stack</p> <p>FLD $ST(i)$ Push $ST(i)$ onto register stack</p>
FLDI	加载+1.0 至寄存器栈。无操作数

(续表)

指 令	描 述
FLDL2T	加载 $\log_2 10$ 至寄存器栈。无操作数
FLDL2E	加载 $\log_2 e$ 至寄存器栈。无操作数
FLDPI	加载 π (圆周率) 至寄存器栈。无操作数
FLDLG2	加载 $\log_{10} 2$ 至寄存器栈。无操作数
FLDLN2	加载 $\log_e 2$ 至寄存器栈。无操作数
FLDZ	加载 +0.0 至寄存器栈。无操作数
FLDCW	加载 16 位内存值至 FPU 控制字中。格式: FLDCW <i>m2byte</i> Load FPU control word from <i>m2byte</i>
FLDENV	从内存中加载 FPU 环境值, 格式: FLDENV <i>m14/28byte</i> Load FPU environment from memory
FMUL	浮点乘法。源操作数和目的操作数相乘并把积存储在目的操作数中。格式: FMUL ST(1) = ST(1) * ST(0), and pop stack FMUL <i>m32fp</i> ST(0) = ST(0) * <i>m32fp</i> FMUL <i>m64fp</i> ST(0) = ST(0) * <i>m64fp</i> FMUL ST(0), ST(i) ST(0) = ST(0) * ST(i) FMUL ST(i), ST(0) ST(i) = ST(i) * ST(0)
FMULP	浮点乘法并出栈。执行的操作与 FMUL 基本相同, 最后要执行出栈操作。格式: FMULP ST(i), ST(0) ST(i) = ST(i) * ST(0), and pop stack
FIMUL	整数转换并相乘。把源整数操作数转换成浮点数并和 ST(0) 相乘, 乘积存储在 ST(0) 中, 格式: FIMUL <i>m16int</i> FIMUL <i>m32int</i>
FNOP	空操作。无操作数
FPATAN	部分反正切。以 $\arctan(ST(1)/ST(0))$ 替代 ST(1) 并出栈。无操作数
FPREM	部分余数。以 ST(0)/ST(1) 的余数替换 ST(0)。无操作数。FPREM1 类似, 以 ST(0)/ST(1) 得到的 IEEE 格式的余数替换 ST(0)
FPTAN	部分正切。以 ST(0) 的正切值替换 ST(0) 并把 1.0 压栈。输入必须是弧度。无操作数
FRNDINT	近似求整。ST(0) 近似到最近的整数值, 无操作数
FRSTOR	恢复 x87 FPU 状态。加载源内存操作数指定的 FPU 状态 (操作环境和寄存器栈)。格式: FRSTOR <i>m94/108byte</i>
FSAVE	存储 x87 FPU 状态。存储当前的 FPU 状态值 (操作环境和寄存器栈) 至目的操作数指定的内存, 然后重新初始化 FPU。格式: FSAVE <i>m94/108byte</i>
FNSAVE	执行同样的操作, 区别在于不会检查未决的未屏蔽异常
FSCALE	执行操作: $ST(0) = ST(0) \times 2^{ST(1)}$, ST(1) 中的值剪切至最近的整数
FSIN	正弦。以 ST(0) 的正弦值替换 ST(0)。输入必须是弧度。无操作数
FSINCOS	正弦和余弦。计算 ST(0) 的正弦和余弦。输入必须是弧度。以 ST(0) 正弦值替换 ST(0), 然后把 ST(0) 的余弦值压栈。无操作数
FSQRT	平方根。以 ST(0) 的平方根替换 ST(0), 无操作数

(续表)

指 令	描 述
FST	<p>存储浮点值。格式：</p> <p>FST <i>m32fp</i> Copy ST(0) to <i>m32fp</i> FST <i>m64fp</i> Copy ST(0) to <i>m64fp</i> FST ST(<i>i</i>) Copy ST(0) to ST(<i>i</i>)</p> <p>FSTP 执行同样的操作，区别在于要执行出栈操作。它有另外一种格式：</p> <p>FSTP <i>m80fp</i> Copy ST(0) to <i>m80fp</i>, and pop stack</p>
FSTCW	<p>存储 FPU 控制字。格式：</p> <p>FLDCW <i>m2byte</i> Store FPU control word to <i>m2byte</i></p> <p>FNSTCW 执行同样的操作，区别在于不检查未决的未屏蔽异常</p>
FSTENV	<p>存储 FPU 环境。根据处理器是在实地址模式下还是保护模式下的不同，在 14 字节或 28 字节的内存结构中存储 FPU 环境信息，格式：</p> <p>FSTENV <i>memop</i> Store FPU environment to <i>memop</i></p> <p>FNSTENV 执行同样操作，区别在于不检查未决的未屏蔽异常</p>
FSTSW	<p>存储 FPU 状态字。格式：</p> <p>FSTSW <i>m2byte</i> Store FPU status word to <i>m2byte</i> FSTSW AX Store FPU status word to AX register</p> <p>FNSTSW 执行同样操作，区别在于不检查未决的未屏蔽异常</p>
FSUB	<p>浮点减法。从目的操作数中减去源操作数并把差存储在目的操作数中。格式：</p> <p>FSUB ST(0) = ST(1) - ST(0), and pop stack FSUB <i>m32fp</i> ST(0) = ST(0) - <i>m32fp</i> FSUB <i>m64fp</i> ST(0) = ST(0) - <i>m64fp</i> FSUB ST(0), ST(<i>i</i>) ST(0) = ST(0) - ST(<i>i</i>) FSUB ST(<i>i</i>), ST(0) ST(<i>i</i>) = ST(<i>i</i>) - ST(0)</p>
FSUBP	<p>浮点减法并出栈。执行与 FSUB 同样的操作，然后再执行出栈操作。格式：</p> <p>FSUBP ST(<i>i</i>), ST(0) ST(<i>i</i>) = ST(<i>i</i>) - ST(0), and pop stack</p>
FISUB	<p>整数转换成浮点数并相减。把整数源操作数转换成浮点数，然后从 ST(0)中减去该值，差存储在 ST(0)中。格式：</p> <p>FISUB <i>m16int</i> ST(0) = ST(0) - <i>m16int</i> FISUB <i>m32int</i> ST(0) = ST(0) - <i>m32int</i></p>
FSUBR	<p>逆向浮点减法。从源操作数中减去目的操作数并把差存储在目的操作数中。格式：</p> <p>FSUBR ST(0) = ST(0) - ST(1), and pop stack FSUBR <i>m32fp</i> ST(0) = <i>m32fp</i> - ST(0) FSUBR <i>m64fp</i> ST(0) = <i>m64fp</i> - ST(0) FSUBR ST(0), ST(<i>i</i>) ST(0) = ST(0) - ST(<i>i</i>) FSUBR ST(<i>i</i>), ST(0) ST(<i>i</i>) = ST(0) - ST(<i>i</i>)</p>
FSUBRP	<p>逆向浮点减法并出栈。首先执行与 FSUBR 同样的操作，然后出栈。格式：</p> <p>FSUBRP ST(<i>i</i>), ST(0) ST(<i>i</i>) = ST(0) - ST(<i>i</i>), and pop stack</p>
FISUBR	<p>整数转换成浮点数并逆向相减。在把整数转换成浮点数之后，执行与 FSUBR 同样的操作。格式：</p> <p>FISUBR <i>m16int</i> FISUBR <i>m32int</i></p>
FTST	<p>测试。比较 ST(0)和 0.0 并相应设置 FPU 状态字中的条件标志。无操作数</p>

(续表)

指 令	描 述
FWAIT	等待。等待所有未决浮点异常处理程序处理完成。无操作数
FXAM	检查。检查 ST(0)并相应设置 FPU 状态字中的条件标志。无操作数
FXCH	交换寄存器的内容。格式： <div>FXCH ST(i) Exchange ST(0) and ST(i) FXCH Exchange ST(0) and ST(1)</div>
FXRSTOR	恢复 x87 FPU, MMX, SSE 和 SSE2 状态。从源操作数指定内存中加载 FPU, MMX, XMM, MXCSR 寄存器。格式： <div>FXRSTOR m512byte</div>
FXSAVE	保存 x87 FPU, MMX, SSE, SSE2 状态。保存 FPU, MMX, XMM, MXCSR 寄存器的当前状态值至目的操作数指定的内存中。格式： <div>FXSAVE m512byte</div>
FXTRACT	析取指数和尾数。把 ST(0)中的源操作数分解成指数和尾数，指数存储在 ST(0)中，尾数压栈。无操作数
FYL2X	计算 $y \times \log_2 x$ 。ST(1)存放 y 的值，ST(0)存放 x 的值，最后要进行出栈操作，因此结果存放在 ST(0)中。无操作数
FYL2XP1	计算 $y \times \log_2 (x + 1)$ ，ST(1)存放 y 的值，ST(0)存放 x 的值，最后要进行出栈操作，因此结果存放在 ST(0)中。无操作数

附录 C BIOS 和 MS-DOS 中断

本章要点

- 简介
- PC 中断
- 中断 21h 的功能调用（MS-DOS 服务）
- 中断 10h 的功能调用（视频 BIOS 服务）
- 键盘 BIOS INT 16h 功能调用
- 鼠标功能调用（INT 33h）

C.1 简介

本附录列出了更多的常用中断，内容共分成下面几组：

- PC 中断号列表。这些中断号对应于存储在内存低端 1024 字节内的中断向量
- INT 21h MS-DOS 功能调用
- INT 10h 视频 BIOS 功能调用
- INT 16h 键盘 BIOS 功能调用
- INT 33h 鼠标功能调用

PC 中断的文档化是一项艰巨的任务，这是由于有许多不同版本的 MS-DOS、各种 DOS 扩展以及多种 PC 硬件控制器。关于中断最好的资源是 Ralf Brown 的 *Interrupt List*，可在英特网上以各种形式获得。我个人最喜欢的是 HTML 的版本，当前版本可在 <http://www.ctyme.com/rbrown.htm> 找到。由于网址经常变更，读者可以通过访问作者的站点获取 Ralf Brown 的 *Interrupt List* 以及其他汇编网站的最新链接。

C.2 PC 中断

表 C.1 PC 中断号列表^①

中断号	描 述
0	除法错误。CPU 产生：当试图除零时发生
1	单步。CPU 产生：当陷阱标志设置时发生
2	不可屏蔽中断。外部硬件：当发生内存错误时产生
3	断点。CPU 产生：当执行机器码 0CCh（INT 3）时发生
4	INTO 检测到除法溢出。CPU 产生：溢出标志设置的条件下执行 INTO 指令时发生
5	打印屏幕。执行 INT 5 或按下 Shift-PrtSc 键时发生
6	无效操作码（80286 以上）
7	处理器扩展不可用（80286 以上）

① 来源：Ray Duncan 的 *Advanced MS-DOS*，第二版，1998 年，Microsoft 出版社，以及 Ralf Brown 的 *Interrupt List*，可在因特网上找到。

(续表)

中断号	描 述
8	IRQ0: 系统时钟中断。更新 BIOS 数据区, 每秒 18.2 次。读者自身的程序需要使用时钟中断时请参见 1Ch
9	IRQ1: 键盘硬件中断。键盘按下时候发生, 从键盘端口读入按键并把按键存储在键盘缓冲区中
0A	IRQ2: 可编程中断控制器
0B	IRQ3: 串行通信口 (COM2)
0C	IRQ4: 串行通信口 (COM1)
0D	IRQ5: 固定硬盘
0E	IRQ6: 磁盘中断。磁盘寻道时发生
0F	IRQ7: 并行打印口
10	视频服务。控制视频显示的例程 (完整列表见表 C.3)
11	设备检查。返回一个字, 显示连接到系统中的所有外围设备
12	内存大小。在 AX 中返回内存的数量 (以 1024 字节的块为单位)
13	软盘服务。包括重置软盘控制器, 获取最近磁盘访问的信息, 读写物理扇区、格式化磁盘等功能
14	异步 (串行) 端口服务。初始化、读写异步通信端口, 返回端口的状态
15	磁带控制器
16	键盘服务。读取或检查键盘输入 (完整列表参见表 C.4)
17	打印机服务。初始化、打印以及返回打印机的状态
18	ROM BASIC。执行 ROM 中固化的 BASIC 解释程序
19	引导加载器。重启 MS-DOS
1A	每天的时间。获取自机器启动时开始的时间计数, 或把该计数设为新值。计数每秒发生 18.2 次
1B	键盘中断。当 Ctrl-Break 按下时候 INT 9h 执行中断处理程序
1C	用户可以使用的时钟中断。空例程, 每秒执行 18.2 次。可由自己的程序使用
1D	视频参数。指向包含视频控制芯片初始化信息的一张表
1E	磁盘参数。指向包含磁盘控制芯片初始化信息的一张表
1F	图形字体表。包含了 8×8 的图形字体, 该表在内存中保留所有 ASCII 码大于 127 的扩展图形字符
20	结束程序。结束一个 COM 程序 (应该尽量使用 INT 21h 4Ch 功能代替该中断)
21	MS-DOS 服务 (完整列表参见表 C.2)
22	MS-DOS 终止地址。指向父程序或进程中的地址。若当前程序结束, 则跳转到该地址
23	MS-DOS 中断地址。在按下 Ctrl-Break 时 MS-DOS 跳转到该地址
24	MS-DOS 关键错误地址。如果当前程序中有严重错误 (比如磁盘介质错误), 则跳转到该地址
25	绝对磁盘读 (已过时)
26	绝对磁盘写 (已过时)
27	结束并驻留 (已过时)

(续表)

中断号	描 述
28~FF	(保留)
33	Microsoft 鼠标。跟踪和控制鼠标的功能调用
34~3E	浮点模拟器
3F	覆盖管理器
40~41	固定磁盘服务。固定磁盘控制器
42~5F	保留：特殊用途
60~6B	应用程序可用
6C~7F	保留：特殊用途
80~F0	保留：由 ROM BASIC 使用
F1~FF	应用程序可用

C.3 中断 21h 的功能调用 (MS-DOS 服务)

中断 21h 在 MS-DOS 服务中有许多功能，在这里仅列出一些最常用的功能。

表 C.2 中断 21h 的功能 (MS-DOS 服务)

功能	描 述
1	从标准输入读取字符。如果没有字符则等待输入。返回：AL = 字符
2	在标准输出上显示字符。接收：DL = 字符
3	从标准辅助输入读取字符 (串口)
4	向标准辅助输出写字符 (串口)
5	向打印机写字符。接收：DL = 字符
6	直接控制台输入/输出。如果 DL = FFh，则从标准输入上读取字符。如果 DL 是其他值，则在标准输出上显示该字符
7	无回显直接字符输入。等待标准输入设备输入一个字符。返回：AL = 字符
8	无回显字符输入。等待标准输入设备输入一个字符。返回：AL = 字符。字符不回显，可以用 Ctrl-Break 结束等待
9	在标准输出上显示字符。接收：DS:DX = 字符串地址
0A	缓冲键盘输入。从标准输入设备读取一个字符串。接收：DS:DX 指向预定义的键盘结构
0B	检查标准输入状态。检查是否有输入字符在等待。返回：如果字符在等待，AL = 0FFh，否则 AL = 0
0C	清除键盘缓冲区并调用输入功能。清除控制台输入缓冲，并执行一个输入功能。接收：AL = 期望的功能 (1, 6, 7, 8 或 0Ah)
0E	选择默认驱动器。接收：DL = 驱动器号 (0 = A, 1 = B, 等)
0F~18	FCB 文件功能调用 (已过时)
19	获取当前的默认驱动器。返回：AL = 驱动器号 (0 = A, 1 = B, 等)
1A	设置磁盘传输地址。接收：DS:DX 包含磁盘传输区域的地址

(续表)

功能	描 述
25	设置中断向量。将中断向量表中的表项设置为新的地址。接收: DS:DX 指向要插入到表中的中断处理程序地址; AL = 中断号
26	创建新的程序段前缀。接收: DX = 新的 PSP 的段地址
27~29	FCB 文件功能调用 (已过时)
2A	获取系统日期。返回: AL = 星期 (0~6, 星期日=0), CX = 年, DH = 月, DL = 日
2B	设置系统日期。接收: CX = 年, DH = 月, DL = 日。返回: 如果日期有效 AL = 0
2C	获取系统时间。返回: CH = 小时, CL = 分钟, DH = 秒, DL = 百秒数
2D	设置系统时间。接收: CH = 小时, CL = 分钟, DH = 秒, DL = 百秒数。返回: 如果时间有效, AL = 0
2E	设置校验标志。接收: AL = MS-DOS 校验标志的新值 (0 = 关闭, 1 = 打开), DL = 00h
2F	获取磁盘传输地址 (DTA)。返回: ES:BX = 地址
30	获取 MS-DOS 的版本号。返回: AL = 主版本号, AH = 次版本号, BH = OEM 序列号, BL: CX = 24 位用户序列号
31	结束并驻留。结束当前的程序或进程, 将其部分留在内存中。接收: AL = 返回码, DX = 请求的小节数
32	获取 MS-DOS 驱动器参数块。接收: DL = 驱动器号。返回: AL = 状态, DS:BX 指向驱动器参数块
33	扩展中断检查。指示 MS-DOS 是否检查 Ctrl-Break
34	获取 INDOS 标志 (未公开)
35	获取中断向量。接收: AL = 中断向量号。返回: ES:BX = 中断处理程序的段/偏移地址
36	获取磁盘的空闲空间 (只用于 FAT16)。接收: DL = 驱动器号 (0 = 默认, 1 = A, 等)。返回: AX = 每簇扇区数, 如果驱动器号无效, AX 返回 FFFFh; BX = 可用的簇数, CX = 每扇区字节数, DX = 每驱动器簇数
37	获取 switch 字符 (命令行参数的前导字符) (未公开)
38	获取或设置国家信息 (细节参见 Duncan 的书或 Brown 的 <i>Interrupt List</i>)
39	创建子目录。接收: DS:DX 指向包含路径和目录名的 ASCIIZ 串。返回: 如果设置了进位标志, AX 返回错误码
3A	删除子目录。接收: DS:DX 指向包含路径和目录名的 ASCIIZ 串。返回: 如果设置了进位标志, AX 返回错误码
3B	改变当前目录。接收: DS:DX 指向包含新路径的 ASCIIZ 串。返回: 如果设置了进位标志, AX 返回错误码
3C	创建或剪裁文件。创建一个新文件或将已存文件剪裁为 0 字节。打开文件用于输出。接收: DS:DX 指向包含文件名的 ASCIIZ 串, CX = 文件属性。返回: 如果设置了进位标志, AX 返回错误码, 否则 AX 返回新的文件句柄
3D	打开已存在的文件。打开文件进行输入、输出或输入输出。接收: DS:DX 指向包含文件名的 ASCIIZ 串, AL = 访问码 (0 = 读, 1 = 写, 2 = 读/写)。返回: 如果发生错误, 设置进位标志并在 AX 中返回错误码, 否则 AX 返回新的文件句柄
3E	关闭文件句柄。关闭文件句柄指定的文件或设备。接收: BX = 前面创建或打开的文件句柄。返回: 如果进位标志设置, 则 AX 返回错误码

(续表)

功能	描 述
3F	读文件或设备。从文件或设备读取指定的字节数。接收: BX = 文件句柄, DS:DX 指向输入缓冲区, CX = 要读的字节数。返回: 如果进位标志设置, AX = 错误码, 否则 AX = 读取的字节数
40	写文件或设备。向文件或设备写指定数目的字节。接收: BX = 文件句柄, DS:DX 指向输入缓冲区, CX = 要写的字节数。返回: 如果进位标志设置, AX = 错误码, 否则 AX = 写入的字节数
41	删除文件。删除指定的文件。接收: DS:DX 指向包含文件名的 ASCII 串。返回: 如果设置进位标志 AX = 错误码
42	移动文件指针。根据指定的方法移动文件的读/写指针。接收: CX:DX = 文件指针的移动距离(字节), AL = 方式码, BX = 文件句柄。方式码如下: 0 = 偏移相对于文件开始, 1 = 偏移相对于当前位置, 2 = 偏移相对于文件的末尾。返回: 如果设置进位标志, AX = 错误码
43	获取/设置文件属性。获取或设置文件的属性。接收: DS:DX = 指向包含文件名的 ASCII 串路径和文件名, CX = 属性, AL = 功能号(1 = 设置属性, 0 = 获取属性)。返回: 如果进位标志设置, AX = 错误码
44	设备的 I/O 控制。获取或设置与打开的文件句柄关联的设备信息, 或者向设备句柄发送一个控制字符串, 或从设备句柄接收一个控制字符串
45	复制文件句柄。为当前打开的文件返回一个新的文件句柄。接收: BX = 文件句柄。返回: 如果进位标志设置, AX = 错误码
46	强制复制文件句柄。强制 CX 内的文件句柄与 BX 内的文件句柄引用同一文件的同一位置。接收: BX = 已存在的文件的句柄, CX = 第二个文件句柄。返回: 如果设置进位标志, AX = 错误码
47	获取当前目录。获取当前目录的全路径名。接收: DS:SI 指向存放目录路径的 64 字节区域, DL = 驱动器号。返回: DS:SI 指向的缓冲区以路径填充, 如果进位标志设置, AX = 错误码
48	分配内存。分配请求的内存, 按小节数(16 字节的块)计算。接收: BX = 请求的小节数。返回: AX = 已分配块所在的段, BX = 可用的最大块(按小节计算), 如果设置了进位标志, AX = 错误码
49	释放已分配的内存块。释放前面使用功能 48h 分配的内存。接收: ES = 要释放块所在的段。返回: 如果进位标志设置, AX = 错误码
4A	修改内存块。修改已分配的内存块的大小。内存块可能会增大或减小。接收: ES = 块所在的段, BX = 请求的小节数。返回: 如果设置进位标志, AX = 错误码, BX = 最多可用块
4B	加载或执行程序。为其他程序创建程序段前缀, 将其加载进内存并执行。接收: DS:DX 指向包含程序的驱动器名、路径和文件名的 ASCII 串, ES:BX 指向参数块, AL = 功能值。AL 内的功能值如下: 0 = 加载并执行程序, 3 = 加载但不执行(覆盖程序)。返回: 如果设置了进位标志, AX = 错误码
4C	结束进程。结束程序返回到 MS-DOS 或调用程序的常用方法。接收: AL = 8 位的返回码, 可以用 MS-DOS 的 4Dh 功能或批处理文件中的 ERRORLEVEL 命令查询

(续表)

功能	描 述
4D	获取进程的返回码。获取进程或程序的返回码, 由 31h 或 4Ch 功能调用产生。返回: AL = 程序的 8 位返回码, AH = 产生退出的类型: 0 = 正常退出, 1 = 按下 Ctrl-Break 退出, 2 = 由关键设备的错误退出, 3 = 调用功能 31h 退出
4E	查找第一个匹配文件。查找与给定文件名匹配的的第一个文件。接收: DS:DX 指向要查找的 ASCIIZ 串文件名; CX = 搜索时使用的文件属性。返回: 如果设置了进位标志, AX = 错误码, 否则当前的 DTA 以文件的名字、属性、时间、日期、大小填充。通常在该功能调用之前要调用 DOS 功能 1Ah (设置 DTA)
4F	查找下一个匹配文件。查找与给定文件名匹配的下一个文件。应该在 DOS 功能 4Eh 调用之后调用该功能。返回: 如果进位标志设置, AX = 错误码, 否则当前的 DTA 填充文件的相关信息
54	获取验证标志。返回: AH = 磁盘 I/O 的验证标志 (0 = 关闭, 1 = 开)
56	重命名/移动文件。重命名文件或将其移动到其他目录中。接收: DS:DX 指向包含当前驱动器名、路径和文件名的 ASCIIZ 串, ES:DI 指向新的路径和文件名。返回: 如果进位标志设置, AX = 错误码
57	获取/设置文件的日期/时间。获取或设置文件的时间、日期戳记。接收: 若要获取日期/时间, AL = 0; 若要设置日期/时间, AL = 1; BX = 文件句柄; CX = 当前文件的时间, DX = 当前文件的日期
58	获取或设置内存分配策略 ^①
59	获取扩展错误信息。返回 MS-DOS 错误的附加信息, 包括错误类别、位置和推荐动作。接收: BX = MS-DOS 版本号 (版本 3.xx 为 0)。返回: AX = 扩展错误码, BH = 错误类别, BL = 建议动作, CH = 位置
5A	创建临时文件。在指定目录内生成一个独一无二的文件。接收: DS:DX 指向以反斜线 (\) 结尾的包含路径名的 ASCIIZ 串; CX = 期望的文件属性。返回: 如果设置了进位标志, AX = 错误码, 否则 DS:DX 指向创建的文件的完整路径
5B	创建新文件。试图创建新文件, 但是如果文件名已经存在则失败, 这防止了覆盖已存在的文件。接收: DS:DX 指向包含路径和文件名的 ASCIIZ 串。返回: 如果设置了进位标志, AX = 错误码
5C~61	(省略)
62	获取程序段前缀 (PSP) 地址。返回: BX = 当前程序段前缀的段值
7303h	获取磁盘剩余空间。填充一个包含磁盘空间详细信息结构。接收: AX = 7303h, ES:DI 指向 ExtGetDskFreSpcStruc 结构, CX = ExtGetDskFreSpcStruc 结构的大小, DS:DX 指向一个包含驱动器名的以 NULL 结尾的字符串。返回: ExtGetDskFreSpcStruc 结构填充了磁盘信息。细节参见 14.5.1 节
7305h	绝对磁盘读写。读写单个或一组磁盘扇区。在 Windows NT/2000/XP 下不能工作。接收: AX = 7305h, DS:BX = DISKIO 结构变量的段/偏移地址, CX = 0FFFFh, DL = 驱动器号 (0 = 默认, 1 = A, 2 = B, 3 = C, 等), SI = 读写标志。细节参见 14.4 节

① 细节请参见 Ray Duncan 的 *Advanced MS-DOS*, 第二版, 1998 年, Microsoft 出版社, 以及 Ralf Brown 的 *Interrupt List*, 可在因特网上找到。

C.4 中断 10h 的功能调用（视频 BIOS 服务）

表 C.3 中断 10h 的功能调用（视频 BIOS 服务）

功能	描 述
0	设置显示模式。把视频显示设为单色、文本、图形或彩色模式。接收：AL = 显示模式
1	设置光标线。设置光标的起始和结束扫描线。接收：CH = 起始线，CL = 结束线
2	设置光标位置。在屏幕上定位光标。接收：BH = 视频页，DH = 行，DL = 列
3	获取光标位置。获取光标的屏幕位置和大小。接收：BH = 视频页。返回：CH = 光标的起始扫描线，CL = 光标的结束扫描线，DH = 行，DL = 列
4	读光笔。读取光笔的位置和状态信息。返回：CH = 像素行，CL = 像素列，DH = 字符行，DL = 字符列
5	设置显示页。选择要显示的视频页。接收：AL = 期望的视频页号
6	上卷窗口。上卷当前视频页上的窗口，空出的行以空白填充。接收：AL = 卷动的行数，BH = 卷动行的属性，CX = 左上角的行列位置，DX = 右下角的行列位置
7	下卷窗口。下卷当前视频页上的窗口，空出的行以空白填充。接收：AL = 卷动的行数，BH = 卷动行的属性，CX = 左上角的行列位置，DX = 右下角的行列位置
8	读取字符及其属性值。读取当前光标位置的字符及其属性值。接收：BH = 显示页。返回：AH = 属性字节，AL = ASCII 码
9	显示指定属性的字符。在当前光标位置显示属性及字符。接收：AL = ASCII 字符，BH = 视频页，BL = 属性，CX = 重复计数
0A	显示字符。在当前光标位置显示字符（无属性字节）。接收：AL = ASCII 字符，BH = 视频页，CX = 重复计数
0B	设置调色板。为 EGA 适配卡的选择一组可用的颜色。接收：BH = 显示方式，BL = 调色板 ID
0C	显示像素。在彩色图形模式下写像素。接收：AL = 像素值，CX = X 坐标，DX = Y 坐标
0D	读取像素。读取指定位置像素点的色彩值。接收：CX = X 坐标，DX = Y 坐标。返回：AL = 像素值
0E	显示字符。在屏幕上显示字符并前进光标。接收：AL = ASCII 字符码，BH = 视频页，BL = 属性或颜色
0F	获取当前视频模式。返回：AL = 视频模式，BH = 活跃的视频页
10	设置视频调色板（只用于 EGA）。设置视频调色板寄存器，边界颜色或闪烁/亮度位。接收：AL = 功能码（00 = 设置调色板寄存器，01 = 设置边界颜色，02 = 设置调色板和边界颜色，03 = 设置/重设亮度位），BH = 颜色，BL = 要设置的调色板寄存器。如果 AL = 2，ES:DX 指向一个颜色列表
11	字符发生器。为 EGA 显示选择字符大小。比如，8×8 字体用于 43 行的显示，8×14 的字体用于 25 行的显示
12	其他选择功能。返回 EGA 显示的技术信息
13	显示字符串（用于 PC/AT）。在视频显示上显示字符串文本。接收：AL = 模式，BH = 页号，BL = 属性，CX = 字符串的长度，DH = 行，DL = 列，ES:BP 指向字符串（在 IBM-PC 或 PC/XT 上不能工作）

C.5 键盘 BIOS INT 16h 的功能调用

表 C.4 键盘 BIOS 中断 16h 功能调用

功能	描 述
03h	设置击键重复速率。接收: AH = 03h, AL = 5, BH = 重复延迟, BL = 重复速率。BH 中的延迟值可以是: 0 = 250 ms, 1 = 500 ms, 2 = 750 ms, 3 = 1000 ms。BL 中的重复速率可以是 0 (最快) 到 1Fh (最慢) 之间的值。返回值: 无
05h	按键送入缓冲区。把键盘字符及其对应的扫描码送入键盘缓冲区。接收: AH = 05h, CH = 扫描码, CL = 字符。如果缓冲区已满, 设置进位标志, AL = 1。返回值: 无
10	等待按键。等待输入字符及其扫描码。接收: AH = 10h。返回: AH = 扫描码, AL = 字符的 ASCII 码 (功能 00h 同 10h 功能相同, 不过只用于老式键盘)
11	检查按键缓冲区。检查是否有字符在键盘缓冲区中等待。接收: AH = 01h。返回: 如果有按键在等待, 按键扫描码在 AH 中返回, 按键的 ASCII 码在 AL 中返回, 清除零标志 (但字符仍在缓冲区中)。如果没有按键等待, 零标志置位 (功能 01h 同 11h 功能相同, 不过只用于老式键盘)
12	获取键盘标志。返回在 RAM 低端存储的键盘标志字节。接收: AH = 12h。返回: 在 AX 中返回键盘标志 (功能 02h 同 12h 功能相同, 不过只用于老式键盘)

C.6 鼠标功能调用 (INT 33h)

INT 33h 鼠标功能在 AX 寄存器中接收功能号。关于这些功能的更多信息, 参见 15.6 节。其他鼠标功能参见表 15.9。

表 C.5 INT 33h 鼠标功能调用

功能	描 述
0000h	重设鼠标并获取鼠标状态。接收: AX = 0000h。重设鼠标并确认其可用。鼠标 (如果找到了) 被定位在屏幕的中央, 其显示页被设为页 0, 指针被设置为隐藏状态, mickeys 到像素的比率 and 速度设为默认值。鼠标的移动范围设置为整个屏幕
0001h	显示鼠标指针。AX = 0001h。返回值: 无。鼠标驱动保留该功能调用次数的计数
0002h	隐藏鼠标指针。AX = 0002h。返回值: 无。在鼠标不可见的时候仍然跟踪其位置
0003h	获取鼠标的位置和状态。AX = 0003h。返回值: BX = 鼠标按钮的状态, CX = X 坐标 (像素), DX = Y 坐标 (像素)
0004h	设置鼠标位置。AX = 0004h, CX = X 坐标 (像素), DX = Y 坐标 (像素)。返回值: 无
0005h	获取按钮的下压信息。接收: AX = 0005h, BX = 按钮 ID (0 = 左键, 1 = 右键, 2 = 中键)。返回: AX = 鼠标按键状态, BX = 按钮下压计数, CX = 最后按下按钮的 X 坐标, DX = 最后按下按钮的 Y 坐标
0006h	获取按钮的释放信息。接收: AX = 0006h, BX = 按钮 ID (0 = 左键, 1 = 右键, 2 = 中键)。返回: AX = 鼠标按键状态, BX = 按钮释放计数, CX = 最后释放按钮的 X 坐标, DX = 最后释放按钮的 Y 坐标
0007h	设置水平限制。接收: AX = 0007h, CX = 最小 X 坐标 (像素), DX = 最大 X 坐标 (像素)。返回值: 无
0008h	设置垂直限制。接收: AX = 0008h, CX = 最小 Y 坐标 (像素), DX = 最大 Y 坐标 (像素)。返回值: 无

附录 D 习题答案

第 1 章 基本概念

1.1 欢迎来到汇编语言的世界

1. 汇编器把汇编语言源程序转换为机器语言。链接器把汇编器创建的一个或多个目标文件组装成可执行程序。
2. 汇编语言是学习应用程序如何通过中断处理、系统调用和共享内存区同计算机操作系统通信的绝好工具。汇编语言程序设计也有助于理解操作系统是如何加载和执行应用程序的。
3. 在一对多的关系中，一条语句扩展成多条汇编语句或机器指令。
4. 一种语言的源程序如果可以在多种计算机系统上编译并运行，则称为是可移植的。
5. 否。每种汇编语言都是基于某类处理器或某种特定的计算机的。
6. 一些嵌入式系统应用的例子：如汽车的燃油控制和点火系统、空调控制系统、安全系统、飞行控制系统、手持计算机、调制解调器、打印机和其他计算机外围智能设备等。
7. 设备驱动是把通用的操作系统命令转换为那些只有设备制造商才知道的硬件操作细节的程序。
8. C++不允许将一种类型的指针赋值给另一种类型的指针，汇编语言对指针没有这样的限制。
9. 适合用汇编语言编写的应用：硬件驱动、嵌入式系统、需要直接访问硬件的计算机游戏。
10. 高级语言未必提供了直接访问硬件的能力，即使提供了也因为要经常使用大量的技巧而导致维护上的困难。
11. 最小的结构支持使程序员需要人工组织大量代码，使各种不同水平的程序员维护现存代码的难度极高。
12. 表达式 $X = (Y * 4) + 3$ 的汇编代码：

```
mov  eax,Y                ; move Y to EAX
mov  ebx,4                ; move 4 to EBX
imul ebx                  ; EAX = EAX * EBX
add  eax,3                ; add 3 to EAX
mov  X,eax                ; move EAX to X
```

1.2 虚拟机的概念

1. 计算机是分层次设计的，每层都表示一个把高层指令集翻译成低层指令集的翻译层。
2. 机器语言涉及到非常底层的细节，并且是由纯粹的数字构成的，难以被人理解。
3. 对。
4. 整个 L1 源程序用一个特别设计的 L0 程序转换成另一个 L0 程序，生成的 L0 程序就可以直接在计算机硬件上执行了。
5. IA-32 处理器的虚拟 86 操作模式模拟最初 IBM 个人计算机使用的 Intel 8086/8088 处理器的体系结构。
6. Java 字节码是一种底层语言，可以在运行时由一个称为 Java 虚拟机 (JVM) 的程序快速执行。

7. 数字逻辑、微结构、指令集体系结构、操作系统、汇编语言、高级语言。
8. 特殊的微结构指令通常是私有的商业秘密。由于一种基本操作需要 3 到 4 条微指令, 因此使用微指令编程是不切实际的。
9. 指令集体系结构。
10. 第 2 层和第 3 层。

1.3 数据的表示方法

1. 最低有效位 (位 0)。
2. 最高有效位 (序号最高的位)。
3. (a) 248 (b) 202 (c) 240
4. (a) 53 (b) 150 (c) 204
5. (a) 00010001 (b) 101000000 (c) 00011110
6. (a) 110001010 (b) 110010110 (c) 100100001
7. (a) 2 (b) 4 (c) 8
8. (a) 16 (b) 32 (c) 64
9. (a) 7 (b) 9 (c) 16
10. (a) 12 (b) 16 (c) 22
11. (a) CF57 (b) 5CAD (c) 93EB
12. (a) 35DA (b) CEA3 (c) FEDB
13. (a) 1110 0101 1011 0110 1010 1110 1101 0111
(b) 1011 0110 1001 0111 1100 0111 1010 0001
(c) 0010 0011 0100 1011 0110 1101 1001 0010
14. (a) 0000 0001 0010 0110 1111 1001 1101 0100
(b) 0110 1010 1100 1101 1111 1010 1001 0101
(c) 1111 0110 1001 1011 1101 1100 0010 1010
15. (a) 58 (b) 447 (c) 16 534
16. (a) 98 (b) 457 (c) 27 227
17. (a) FFE6 (b) FE3C
18. (a) FFE0 (b) FFC2
19. (a) +31 915 (b) -16 093
20. (a) +32 667 (b) -32 208
21. (a) -75 (b) -42 (c) -16
22. (a) -128 (b) -52 (c) -73
23. (a) 11111011 (b) 11011100 (c) 11110000
24. (a) 10111000 (b) 10011110 (c) 11100110
25. 58h 和 88d。
26. 4Dh 和 77d。
27. 是为了处理那些包含多于 256 个字符代码的国际字符集。
28. $2^{256} - 1$
29. $-2^{255} - 1$

1.4 布尔运算

1. $(\text{NOT } X) \text{ OR } Y$ 。
2. $X \text{ AND } Y$ 。
3. T。
4. F。
5. T。
6. 真值表:

A	B	$A \vee B$	$\neg(A \vee B)$
F	F	F	T
F	T	T	F
T	F	T	F
T	T	T	F

7. 真值表:

A	B	$\neg A$	$\neg B$	$\neg(A \wedge \neg B)$
F	F	T	T	T
F	T	T	F	F
T	F	F	T	F
T	T	F	F	F

8. 16 或 2^4 。
9. 2 位, 得到下面的值: 00, 01, 10, 11。

第2章 IA-32 处理器体系结构

2.1 基本概念

1. 控制单元、算术逻辑单元和时钟。
2. 数据总线、地址总线和控制总线。
3. 常规内存在 CPU 的外部并且响应访问请求的速度比较慢, 寄存器在 CPU 内同其他部件是直接相连的, 所以访问较快。
4. 取指、解码、执行。
5. 取内存操作数、存储内存操作数。
6. 取指令阶段。
7. 并行执行处理器的各个阶段, 使得交迭执行机器指令成为可能。
8. 10 个时钟周期。
9. 12 个时钟周期: $5 + (8 - 1)$ 。
10. 超标量处理器是包含两个以上执行流水线的处理器。
11. 15 个时钟周期: $5 + 10$ 。
12. 2.1.4 节提到了文件名、文件大小以及在磁盘上的起始位置。(大多数目录都存储了文件的

最后修改的日期和时间。)

13. 操作系统执行一条分支 (就像一条 GOTO 语句) 指令, 跳转到程序的第一条机器指令处执行。
14. CPU 通过在程序之间快速切换执行多个任务 (程序), 这给人以多个程序同时运行的假象。
15. 操作系统调度器决定了要分配给每个任务多少时间并切换任务。
16. 程序计数器、任务的变量以及 CPU 的寄存器 (包括状态标志)。
17. 3.33×10^{-10} , 也就是 $1.0/3.0 \times 10^{-9}$ 。

2.2 IA-32 处理器体系结构

1. 实地址模式、保护模式和系统管理模式。
2. EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP。
3. CS, DS, SS, ES, FS, GS。
4. 循环计数器。
5. EBP。
6. 最常用的: 进位标志、符号标志、零标志、溢出标志。不太常用的: 辅助进位标志、奇偶标志。
7. 进位标志。
8. 溢出标志。
9. 符号标志。
10. 浮点单元。
11. 80 位。
12. Intel 80386。
13. 奔腾 (Pentium)。
14. 奔腾 II (Pentium II)。
15. CISC 的含义是复杂指令集: 包含非常多的指令, 其中的一些指令执行类似于高级语言指令的非常复杂的操作。
16. RISC 代表精简指令集: 包含很少的简单 (原子) 的指令, 这些指令可以组合起来完成更加复杂的操作。

2.3 IA-32 的内存管理

1. 4 GB (0~FFFFFFFFh)。
2. 1 MB (0~FFFFFh)。
3. 线性 (绝对) 地址。
4. 09600h。
5. 0CFF0h。
6. 32 位。
7. SS 寄存器。
8. 局部描述符表。
9. 全局描述符表。
10. 装入内存的所有程序的总的大小可以超过计算机安装的物理内存的大小。
11. 这是一个自由回答问题。事实是 MS-DOS 最初必须在 8086/8088 处理器上运行, 而 8086/8088

处理器只支持实地址模式。在新的处理器出现之后,关于 MS-DOS 仍然不支持保护模式的原因,我的猜测是 Microsoft 仍然想让 MS-DOS 在老处理器上运行,否则老计算机用户将拒绝升级到新版本的 MS-DOS。

12. 下面的段-偏移地址指向同样的线性地址: 0640:0100 和 0630:0200。

2.4 IA-32 微机的构成

1. SRAM 是静态 RAM,用于 CPU 的缓存。
2. 奔腾 (Pentium)。
3. 8259 是中断控制芯片,有时也称为可编程中断控制器 (PIC),调度硬件中断并向 CPU 发送中断信号。
4. 在视频显示卡的内部或主板上 (特殊的内存区域)。
5. 电子束照亮屏幕上的称为像素的荧光点,电子枪从屏幕的最顶端开始从左到右进行扫描,然后关闭,并重新从左边的下一新行开始扫描,水平回扫指电子枪在行扫描之间关闭的时间。当绘制完最后一行的时候,电子枪关闭并移动到屏幕的左上角重新开始扫描 (称为垂直回扫)。
6. 动态 RAM、静态 RAM、视频 RAM 和 CMOS RAM。
7. 静态 RAM。
8. 计算机可查询通过 USB 连接的设备的名字、设备类型以及它所支持的驱动的类型,计算机还可以挂起单个 USB 设备的电源。这些功能对于串口或并口而言都是不可能的。
9. 传入接头和传出接头。
10. 16550 UART (通用异步收发器, Universal Asynchronous Receiver Transmitter)。

2.5 输入输出系统

1. 应用程序层。
2. BIOS 功能调用直接同系统硬件通信,和操作系统无关。
3. 总会发明新的设备,在编写 BIOS 时不能预见到并支持这些新功能,因此设备驱动是必需的。
4. BIOS 层。
5. 操作系统层、BIOS 层和硬件层。
6. 游戏程序通常试图利用特定声卡的最新特性。应该注意的是 MS-DOS 下的应用程序在这一点上比在 MS-Windows 下运行的游戏程序做起来更容易一些,Windows NT/2000/XP 禁止应用程序直接访问硬件。
7. 不能。同样的 BIOS 对于不同的操作系统应都能工作,许多计算机用户在同一计算机上安装两个以上的操作系统,他们绝对不愿意在每次重启计算机后修改系统的 BIOS!

第 3 章 汇编语言基础

3.1 汇编语言的基本元素

1. h, q, o, d, b, r, t, y。
2. 否 (要求以 0 开头)。
3. 否 (它们的优先级相同)。
4. 表达式: $10 \bmod 3$ 。
5. 实数常量: $+3.5E-02$ 。
6. 否。也可以使用双引号括起。

7. 伪指令。
8. 247 个字符。
9. 对。
10. 对。
11. 错。
12. 对。
13. 标号、助记符、操作数和注释。
14. 对。
15. 对。
16. 代码示例:

```
Comment !  
    This is a comment  
    This is also a comment  
!
```

17. 这是由于在指令使用的当前变量之前插入新的变量时编码在指令中地址都要重新修改。

3.2 例子: 整数相加和减

1. INCLUDE 伪指令从 Irvine32.inc 文件中复制必需的定义和设置信息, 该文件的数据流将被插入到汇编器读取的数据流中。
2. .CODE 伪指令标记代码段的开始。
3. 代码段、数据段和堆栈段。
4. 调用 DumpRegs 过程。
5. exit 语句。
6. PROC 伪指令。
7. ENDP 伪指令。
8. END 伪指令标记被汇编程序的结束, END 后的标号标识了程序的入口点 (执行开始的地方)。
9. PROTO 伪指令声明了当前程序调用的过程的名字。

3.3 汇编、链接和运行程序

1. 对象文件 (.OBJ) 和列表文件 (.LST)。
2. 对。
3. 对。
4. 加载器。
5. 可执行程序 (.EXE) 和映射文件 (.MAP)。

3.4 定义数据

1. var1 SWORD。
2. var2 BYTE。
3. var3 SBYTE。
4. var4 QWORD。
5. SDWORD。
6. var5 SDWORD - 2 147 483 648。

7. wArray WORD 10, 20, 30。
8. myColor BYTE "blue", 0。
9. dArray DWORD 50 DUP(?)。
10. myTestString BYTE 500 DUP("TEST")。
11. bArray BYTE 20 DUP(0)。
12. 21h, 43h, 65h, 87h。

3.5 符号常量

1. BACKSPACE = 08h。
2. SecondsInDay = 24 * 60 * 60。
3. ArraySize = (\$ - myArray)。
4. ArraySize = (\$ - myArray) / TYPE DWORD。
5. PROCEDURE TEXTEQU <PROC>。
6. 代码示例:

```
Sample TEXTEQU <"This is a string">
MyString BYTE Sample
```

7. SetupESI TEXTEQU <mov esi, OFFSET myArray>。

第4章 数据传送、寻址和算术运算

4.1 数据传送指令

1. 寄存器、立即数和内存操作数。
2. 错。
3. 错。
4. 对。
5. 32位的寄存器或内存操作数。
6. 16位的立即数(常量)操作数。
7. (a) 无效 (b) 有效 (c) 无效 (d) 无效
(e) 无效 (f) 无效 (g) 有效 (h) 无效
8. (a) FCh (b) 01h
9. (a) 1000h (b) 3000h (c) FFF0h (d) 4000h
10. (a) 00000001h (b) 00001000h (c) 00000002h (d) FFFFFFFCh

4.2 加法和减法

1. inc val2。
2. sub eax, val3。
3. 代码:
mov ax, val4
sub val2, ax
4. CF = 0, SF = 1。
5. OF = 1, SF = 1。
6. 写下面的标志值:
(1) CF = 1, SF = 0, ZF = 1, OF = 0
(2) CF = 0, SF = 1, ZF = 0, OF = 1
(3) CF = 0, SF = 1, ZF = 0, OF = 1

7. 代码示例:

```

mov ax, val2
neg ax
add ax, bx
sub ax, val4

```

8. 否。

9. 是。

10. 是 (例如, `mov al, -128` 后面跟 `neg al`)。

11. 否。

12. 下面的代码同时设置进位和溢出标志:

```

mov al, 80h
add al, 80h

```

13. 下面的代码在 INC 和 DEC 之后设置零标志, 表明发生了无符号溢出:

```

mov al, 0FFh
inc al
jz overflow_occurred
mov bl, 1
dec bl
jz overflow_occurred

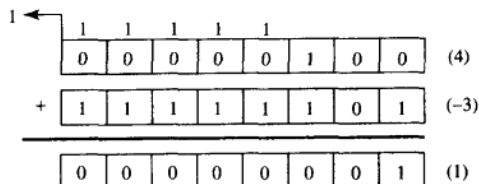
```

14. 从 4 中减去 3 (无符号), MSB 到高位的进位被放在进位标志中了:

```

mov al, 4
sub al, 3      ; CF = 0

```



4.3 和数据相关的操作符和伪指令

1. 错。

2. 错。

3. 对。

4. 错。

5. 对。

6. 数据定义伪指令:

```

.data
ALIGN 2
myBytes BYTE 10h, 20h, 30h, 40h
etc.

```

7. (a) 1 (b) 4 (c) 4 (d) 2 (e) 4 (f) 8 (g) 5

8. `mov dx, WORD PTR myBytes。`9. `mov al, BYTE PTR myWords + 1。`10. `mov eax, DWORD PTR myBytes。`

11. 数据定义伪指令:

```

myWordsD LABEL DWORD
myWords WORD 3 DUP(?),2000h
.data
mov eax,myWordsD

```

12. 数据定义伪指令:

```

myBytesW LABEL WORD
myBytes BYTE 10h,20h,30h,40h
.code
mov ax,myBytesW

```

4.4 间接寻址

1. 错。
2. 对。
3. 错。
4. 错。
5. 对 (需要 PTR 操作符)。
6. 对。
7. (a) 10h (b) 40h (c) 003Bh (d) 3 (e) 3 (f) 2
8. (a) 2010h (b) 003B008Ah (c) 0 (d) 0 (e) 0044h

4.5 JMP 和 LOOP 指令

1. 对。
2. 错。
3. 4 294 967 296 次。
4. 错。
5. 对。
6. CX。
7. ECX。
8. 错 (距当前位置-128~+127 字节)。
9. 程序不会终止, 由于第一条 LOOP 指令对 ECX 减 1 使得 ECX 等于 0, 第二条 LOOP 指令对 ECX 减 1 导致 ECX 等于 FFFFFFFh, 将导致外层循环继续执行。
10. 在标号 L1 处插入指令: push ecx, 在第二条 LOOP 指令之前插入指令: pop ecx。(一旦插入了这两条指令, EAX 中的最终值是 1Ch。)

第 5 章 过程

5.1 简介

无习题。

5.2 与外部库链接

1. 错。包含了目标代码。
2. 代码示例:
MyProc PROTO
3. 代码示例:
call MyProc
4. Irvine32.lib。
5. Kernel32.lib。
6. Kernel32.dll 是一个动态链接库文件, 它是 MS-Windows 操作系统的基本组件之一。

5.3 本书附带的链接库

1. RandomRange 过程。
2. WaitMsg 过程。
3. 代码示例:

```
mov eax,700
call Delay
```
4. WriteDec 过程。
5. Gotoxy 过程。
6. INCLUDE Irvine32.inc。
7. PROTO 语句 (过程原型声明) 和常量定义语句 (也是文本宏, 但本章没有讲解)。
8. ESI 包含数据的起始地址, ECX 包含数据单元的个数, EBX 包含数据单元的大小 (字节、字或双字)。
9. EDX 包含字节数组的偏移地址, ECX 包含最多要读取的字符数。
10. 进位标志、符号标志、零标志、溢出标志、辅助进位标志和奇偶标志。
11. 代码示例:

```
.data
str1 BYTE "Enter identification number: ",0
idStr BYTE 15 DUP(?)
.code
mov edx,OFFSET str1
call WriteString
mov edx,OFFSET idStr
mov ecx,(SIZEOF idStr) - 1
call ReadString
```

5.4 堆栈操作

1. SS 和 ESP。
2. 运行时栈是惟一由 CPU 直接管理的栈类型。例如, 它可用于存放被调用过程的返回地址。
3. LIFO 表示“后进先出”, 最后压入栈的值是最先从栈中弹出的值。
4. ESP 减 4。
5. 对。
6. 错 (既可以压入 16 位的数值, 也可以压入 32 位的数值)。
7. 对。
8. 错 (80186 以后的处理器是可以的)。
9. PUSHAD。
10. PUSHFD。
11. POPFD。
12. NASM 的方法允许程序员特别指定要压入哪些寄存器, 而 PUSHAD 就没有这种灵活性了。这在一个过程需要同时保存几个寄存器又要通过 EAX 寄存器向调用过程返回值的时候是非常重要的。在这种情况下 EAX 不能压栈和出栈, 否则返回值将会丢失。
13. 和 PUSH EAX 等价的代码。

```
sub esp,4
mov [esp],eax
```

5.5 过程的定义和使用

1. 对。
2. 错。
3. CPU 将会在过程结束之后的地方继续执行, 有可能进入另外一个过程。这种类型的编程

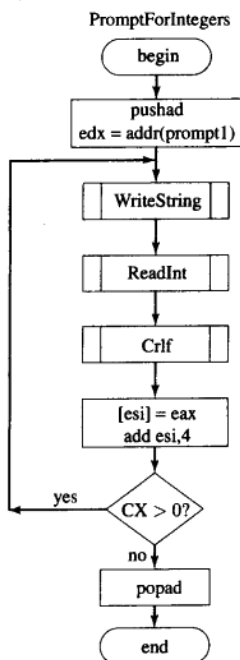
bug 有时非常难以检测。

4. Receives 表示在过程被调用时给它的输入参数。Returns 表示过程返回时可能的返回值。
5. 错 (它压入紧跟在 CALL 指令之后的指令的偏移地址)。
6. 对。
7. 对。
8. 错 (没有 NESTED 操作符)。
9. 对。
10. 错。
11. 对 (它还接受数组元素数)。
12. 对。
13. 错。
14. 错。
15. 下面的语句需要修改:

add eax,[esi] 修改成: add ax,[esi]
add esi,4 修改成: add esi,2

5.6 使用过程进行程序设计

1. 功能分解或自顶向下设计。
2. Clrscr, WriteString, ReadInt, WriteInt。
3. 框架程序包含了所有重要的过程,但是过程为空或接近为空。
4. 错 (接收一个数组的指针)。
5. 下面的语句需要修改:
mov [esi],eax 修改成: mov [esi],ax
add esi,4 修改成: add esi,2
6. PromptForIntegers 过程的流程图:



第6章 条件处理

6.1 简介

无习题。

6.2 布尔和比较指令

1. (a) 00101101 (b) 01001000 (c) 01101111 (d) 10100011
2. (a) 85h (b) 34h (c) BFh (d) AEh
3. (a) CF = 0, ZF = 0, SF = 0。
(b) CF = 0, ZF = 0, SF = 0。
(c) CF = 1, ZF = 0, SF = 1。
4. and ax,00FFh
5. or ax,0FF00h
6. xor eax,0FFFFFFFh
7. test eax,1 ; (如果 eax 为奇则低位置位)
8. or al,00100000b
9. and al,00001111b
10. 代码示例:

```
.data
memVal DWORD ?
.code
mov al,BYTE PTR memVal
xor al,BYTE PTR memVal+1
xor al,BYTE PTR memVal+2
xor al,BYTE PTR memVal+3
```

6.3 条件跳转

1. JA, JNBE, JAE, JNB, JNAE, JBE, JNA。
2. JG, JNLE, JGE, JNL, JL, JNGE, JLE, JNG。
3. JECXZ。
4. 是。
5. 否 (JB 使用无符号操作数而 JL 使用有符号操作数)。
6. JBE。
7. JL。
8. 否 (8109h 是负数, 26h 是正数)。
9. 是。
10. 是 (-42 的无符号表示和 26 比较)。
11. 代码:

```
cmp dx,cx
jbe L1
```
12. 代码:

```
cmp ax,cx
jg L2
```
13. 代码:

```
and al,11111100b
jz L3
jmp L4
```

14. 三指令序列中的 XOR 指令总是清除进位标志。根据信号量中值的不同, BTC 指令有可能清除进位标志也有可能不清除进位标志。

6.4 条件循环指令

1. 错。
2. 对。
3. 对。
4. 代码示例:

```
.data
array SWORD 3,5,14,-3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
main PROC
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
next:
    test WORD PTR [esi],8000h      ; test sign bit
    pushfd                        ; push flags on stack
    add esi,TYPE array
    popfd                         ; pop flags from stack
    loopz next                    ; continue loop while
ZF=1
    jz quit                      ; none found
    sub esi,TYPE array           ; ESI points to value
```

5. 如果未找到匹配值, ESI 将指向数组的末尾的后一个位置。如果使用 ESI 寻址并修改数据, 有可能导致数据损坏。

6.5 条件结构

本节中假设所有的值都是无符号的。

1. 代码示例:

```
cmp bx,cx
jna next
mov X,1
next:
```

2. 代码示例:

```
cmp dx,cx
jnbe L1
mov X,1
jmp next
L1: mov X,2
next:
```

3. 代码示例:

```
cmp val1,cx
jna L1
cmp cx,dx
jna L1
mov X,1
jmp next
L1: mov X,2
next:
```

4. 代码示例:

```

    cmp bx,cx
    ja  L1
    cmp bx,vall
    ja  L1
    mov X,2
    jmp next
L1:  mov X,1
next:

```

5. 代码示例:

```

    cmp bx,cx                ; bx > cx?
    jna L1                  ; no: try condition after OR
    cmp bx,dx                ; yes: is bx > dx?
    jna L1                  ; no: try condition after OR
    jmp L2                  ; yes: set X to 1
;-----OR(dx > ax) -----
L1:  cmp dx,ax              ; dx > ax?
    jna L3                  ; no: set X to 2
L2:  mov X,1                ; yes:set X to 1
    jmp next                ; and quit
L3:  mov X,2                ; set X to 2
next:

```

6. 未来对该表的修改有可能会改变 NumberOfEntries 的值, 我们有可能会忘记手动更新常量值, 但是汇编器可以正确调整该值。

7. 代码示例:

```

.data
sum DWORD 0
sample DWORD 50
array DWORD 10,60,20,33,72,89,45,65,72,18
ArraySize = ($ - Array) / TYPE array

.code
    mov     eax,0            ; sum
    mov     edx,sample
    mov     esi,0           ; index
    mov     ecx,ArraySize

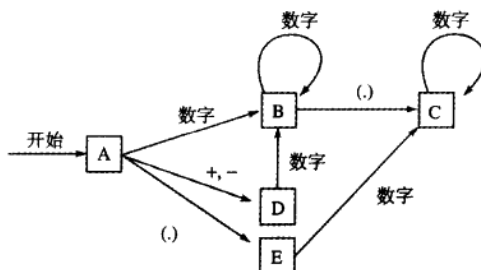
L1:  cmp     esi,ecx
    jnl     L5
    cmp     array[esi*4],edx
    jng     L4
    add     eax,array[esi*4]
L4:  inc     esi
    jmp     L1
L5:  mov     sum,eax

```

6.6 应用: 有限状态机

1. 一个有向图。
2. 每个节点是一个状态。
3. 每条边是由输入导致的从一个状态到另一个状态的转换。
4. 状态 C。
5. 无限个位的数字。
6. FSM 进入一个错误状态。

7. 否。建议的 FSM 允许一个有符号整数仅由一个加号或减号构成，而 6.6.2 节的 FSM 不允许这样做。
8. 识别无指数实数的 FSM:



6.7 决策伪指令 (可选)

无习题。

第7章 整数算术指令

7.1 简介

无习题。

7.2 移位和循环移位指令

1. ROL。
2. RCR。
3. SAR。
4. RCL。
5. 代码示例:

```

shr al,1          ; shift AL into Carry flag
jnc next          ; Carry flag set?
or al,80h         ; yes: set highest bit
next:             ; no: do nothing

```

6. 进位标志接收 AX (在移位之前) 的最低位。

7. shl eax,4
8. shr ebx,2
9. ror dl,4 (或: rol dl,4)
10. shld dx,ax,1

11. (a) 6Ah (b) EAh (c) FDh (d) A9h
12. (a) 9Ah (b) 6Ah (c) 0A9h (d) 3Ah

13. 代码示例:

```

shr ax,1          ; shift AX into Carry flag
rcr bx,1          ; shift Carry flag into BX
; Using SHRD:
shrd bx,ax,1

```

14. 代码示例:


```

        mov ecx,32                ; loop counter
        mov bl,0                  ; counts the '1' bits
L1:     shr eax,1                  ; shift into Carry flag
        jnc L2                    ; Carry flag set?
        inc bl                    ; yes: add to bit count
L2:     loop L1                   ; continue loop

; if BL is odd, clear the parity flag
; if BL is even, set the parity flag
        shr bl,1
        jc odd
        mov bh,0
        or bh,0                  ; PF = 1
        jmp next
odd:
        mov bh,1
        or bh,1                  ; PF = 0
next:

```

7.3 移位和循环移位的应用

1. 这个问题要求从最高位的字节开始，直到最低位的字节：

```

byteArray BYTE 81h,20h,33h
.code
shr byteArray+2,1
rcr byteArray+1,1
rcr byteArray,1

```

2. 这个问题要求从最低位的字开始，直到最高位的字：

```

wordArray WORD 810Dh,0C064h,93ABh
.code
shl wordArray,1
rcr wordArray+2,1
rcr wordArray+4,1

```

3. 乘数 (24) 可分解成 16 + 8:

```

mov ebx,eax                ; save a copy of eax
shl eax,4                  ; multiply by 16
shl ebx,3                  ; multiply by 8
add eax,ebx                ; add the products

```

4. 正如提示所解释的，乘数 (21) 可分解成 16 + 4 + 1:

```

mov ebx,eax                ; save a copy of eax
mov ecx,eax                ; save another copy of eax
shl eax,4                  ; multiply by 16
shl ebx,2                  ; multiply by 4
add eax,ebx                ; add the products
add eax,ecx                ; add original value of eax

```

5. 修改标号 L1 处的指令为 shr eax, 1。

6. 假设时间戳记字在 DX 寄存器中：

```

shr dx,5
and d1,00111111b          ; (leading zeros optional)
mov bMinutes,d1           ; save in variable

```

7.4 乘法和除法指令

1. 乘积存储在两倍于乘数和被乘数大小的寄存器中。如果是 FFh 乘以 FFh，乘积 (FE01h) 可以很容易地存储在 16 个数据位中。

2. 在乘积完全填满低半部分的寄存器时, IMUL 指令对乘积的低半部分所在的寄存器进行符号扩展至乘积的高半部分所在的寄存器; MUL 指令则对乘积的低半部分进行零扩展至乘积的高半部分所在的寄存器。
3. 在乘积的高半部分不是低半部分的符号扩展时, IMUL 指令设置进位标志和溢出标志
4. EAX。
5. AX。
6. AX。
7. 代码示例:

```
mov ax,dividendLow
cwd                      ; sign-extend dividend
mov bx,divisor
idiv bx
```

8. DX = 0002h, AX = 2200h。
9. AX = 0306h。
10. EDX = 0, EAX = 00012340h。
11. DIV 指令将导致除法溢出, 因此 AX 和 DX 的值是不确定的。
12. 代码示例:

```
mov ax,3
mov bx,-5
imul bx
mov val1,ax              ; product
// alternative solution:
mov al,3
mov bl,-5
imul bl
mov val1,ax              ; product
```

13. 代码示例:

```
mov ax,-276
cwd                      ; sign-extend AX into DX
mov bx,10
idiv bx
mov val1,ax              ; quotient
```

14. 算术表达式 $val1 = (val2 * val3) / (val4 - 3)$ 的实现:

```
mov eax,val2
mul val3
mov ebx,val4
sub ebx,3
div ebx
mov val1,ebx
```

(在这个例子中可把 EBX 替换为任意的 32 位通用寄存器。)

15. 算术表达式 $val1 = (val2 / val3) * (val1 + val2)$ 的实现:

```
mov eax,val2
cdq                      ; extend EAX into EDX
idiv val3                 ; EAX = quotient
mov ebx,val1
add ebx,val2
imul ebx
mov val1,ebx              ; lower 32 bits of product
```

(在这个例子中可把 EBX 替换为任意的 32 位通用寄存器。)

7.5 扩展加法和减法

1. ADC 指令把源操作数、目的操作数以及进位标志相加。
2. SBB 指令从目的操作数中减去源操作数和进位标志的值。
3. EAX = C0000000h, EDX = 00000010h。
4. EAX = F0000000h, EDX = 000000FFh。
5. DX = 0016h。
6. 在修正这个例子时,减少指令的数目是最容易的。可以使用一个寄存器 (ESI) 寻址所有三个变量。在循环开始之前 ESI 应设为 0,这是由于整数是以小尾顺序 (低位字节出现在前面) 存储的:

```

        mov ecx,8                ; loop counter
        mov esi,0                ; use the same index reg
        cld                      ; clear Carry flag
top:
        mov al,byte ptr val1[esi] ; get first number
        sbb al,byte ptr val2[esi] ; subtract second
        mov byte ptr result[esi],al ; store the result
        inc esi                  ; move to next pair
        loop top

```

当然,也可以通过增加双字而不是字节的方法轻易地减少循环迭代的次数。

7.6 ASCII 和未压缩十进制算术指令

1. 代码示例:

```
or ax,3030h
```

2. 代码示例:

```
and ax,0F0Fh
```

3. 代码示例:

```
and ax,0F0Fh                ; convert to unpacked
aad
```

4. 代码示例:

```
aam
```

5. 例子代码 (显示 AX 中的二进制值):

```

out16 PROC
    aam
    or     ax,3030h
    push  eax
    mov    al,ah
    call  WriteChar
    pop    eax
    call  WriteChar
    ret
out16 ENDP

```

6. 在 AAA 指令执行后,AX 等于 0108h。Intel 说:如果 AL 的低 4 位大于 9 或辅助进位标志置位,AL 加 6,AH 加 1。然后无论在何种情况下,AL 都要和 0Fh 进行与操作,伪代码如下:

```

IF ((AL AND 0FH) > 9) OR (AuxCarry = 1) THEN
    add 6 to AL
    add 1 to AH
END IF
AND AL with 0FH;

```

7.7 压缩十进制算术指令

1. 压缩十进制加法之和大于 99 时, DAA 设置进位标志, 例如:

```

mov al, 56h
add al, 92h                ; AL = E8h
daa                       ; AL = 48h, CF=1

```

2. 在较小的压缩十进制数减去较大的压缩十进制数时, DAS 设置进位标志, 例如:

```

mov al, 56h
sub al, 92h                ; AL = C4h
das                       ; AL = 64h, CF=1

```

3. $n+1$ 个字节。
4. 假设 $AL = 3Dh$, $AF = 0$ 并且 $CF = 0$, 由于低 4 位 (D) 大于 9, 因此低位的数字 D 减去 6, AL 现在等于 $37h$ 。由于高 4 位 (3) 小于等于 9 并且 $CF = 0$, 因此无需其他调整, DAS 产生结果 $AL = 37h$ 。

第 8 章 高级过程

8.1 简介

无习题。

8.2 堆栈框架

- 对。
- 对。
- 对。
- 错。
- 对。
- 对。
- 传值参数和传引用参数。
- 代码示例:


```

mov esp, ebp
pop ebp

```
- EAX。
- 使用 STDCALL 调用约定的过程 RET 指令后跟一个常量。在 RET 指令从堆栈中弹出过程的返回地址后, 堆栈指针值要和该常量相加。
- 堆栈框架图:

10h	[EBP + 16]
20h	[EBP + 12]
30h	[EBP + 8]
(返回地址)	[EBP + 4]
EBP	<--ESP

12. LEA 可以返回间接操作数的偏移地址,这在获取堆栈参数的偏移地址时特别有用。

13. 4 个字节。

14. 代码示例:

```
AddThree PROC
; modeled after the AddTwo procedure in Section 8.4.3:
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 16]; 10h
    add  eax,[ebp + 12]; 20h
    add  eax,[ebp + 8] ; 30h
    pop  ebp
    ret  12
AddThree ENDP
```

15. 字符零扩展至 EAX 并压入堆栈。

16. 声明: LOCAL pArray:PTR DWORD

17. 声明: LOCAL buffer[20]:BYTE

18. 声明: LOCAL pwArray:PTR WORD

19. 声明: LOCAL myByte:SBYTE

20. 声明: LOCAL myArray[20]:DWORD

21. C 调用约定规定由调用程序清理堆栈,因此使用 C 调用约定可以创建可变参数的过程或函数。

8.3 递归

1. 错。

2. n 等于 0 时终止。

3. 在每次递归调用完成之后执行下面的指令:

```
ReturnFact:
    mov  ebx,[ebp+8]
    mul  ebx
L2:    pop  ebp
    ret  4
```

4. 已计算的值已经超出了一个无符号双字所能表示的值范围并发生回滚,输出看起来似乎比 12 的阶乘要小。

5. 计算 12! 时使用 156 字节的堆栈空间。原理:从图 8.1 中可以看到,当 $n=0$ 时,使用 12 个堆栈字节(3 个堆栈项);当 $n=1$ 时,使用 24 个堆栈字节;当 $n=2$ 时,使用 36 个堆栈字节。依此类推,计算 $n!$ 需要的堆栈空间是 $(n+1) \times 12$ 。

6. 计算斐波那契数列时使用递归算法对系统资源使用的效率很低,这是由于每次以 n 为参数调用递归的斐波那契数计算过程时都会递归调用该过程计算 $1 \sim n-1$ 之间的斐波那契数。下面是生成斐波那契数列前 20 个值的伪码:

```
for(int i = 1; i <= 20; i++)
    print( fibonacci(i) );
int fibonacci(int n)
{
    if( n == 1 )
        return 1;
```

```

        elseif( n == 2 )
            return 2;
        else
            return fibonacci(n-1) + fibonacci(n-2);
    }

```

8.4 .MODEL 伪指令

1. 小型内存模式只包含一个代码段和一个数据段，所有的代码和数据都是近程的，也就是说只能使用 16 位偏移地址访问。
2. 平坦内存模式只能用于保护模式，所有的偏移地址都是 32 位的，代码和数据都属于同一个段。
3. 使用 C 关键字的程序由调用过程负责清理堆栈，使用 STDCALL 关键字的程序由被调用过程负责清理堆栈。

8.5 INVOKE, ADDR, PROC 和 PROTO

1. 对。
2. 错。
3. 错。
4. 对。
5. 错。
6. 对。
7. 对。
8. 声明：

```

MultArray PROC ptr1:PTR DWORD,
               ptr2:PTR DWORD,
               count:DWORD
               ; (may be byte, word, or dword)

```

9. 声明：

```

MultArray PROTO ptr1:PTR DWORD,
                  ptr2:PTR DWORD,
                  count:DWORD
                  ; (may be byte, word, or dword)

```

10. 输入输出参数。
11. 输出参数。

8.6 创建多模块程序

1. 对。
2. 错。
3. 对。
4. 错。

第 9 章 字符串和数组

9.1 简介

无习题。

9.2 基本字符串操作指令

1. EAX。
2. SCASD。

3. (E)DI。
4. LODSW。
5. 在 $ZF = 0$ 时重复指令。
6. 1 (置位)。
7. 2。
8. 不管 CMPS 使用什么操作数, 指令总是比较 ESI 指向的内存地址和 EDI 指向的内存地址处的内容。
9. 匹配字符之后的一个字节。
10. REPNE (REPZ)。

9.3 精选的字符串过程

1. 错 (在到达较短字符串结尾的空字符处停止)。
2. 对。
3. 错。
4. 错。
5. 1 (置位)。
6. 检查字符串是否只包含要删除的字符。
7. 数字不会改变。
8. REPNE (REPZ)。
9. 长度将等于 (最终的 EDI 值 - 初始的 EDI 值) - 1。

9.4 二维数组

1. 任意 32 位通用寄存器。
2. $[ebx + esi]$ 。
3. $array[ebx + esi]$ 。
4. 16。
5. 代码示例:

```
mov esi, 2           ; row
mov edi, 3           ; column
mov eax, [esi*16 + edi*4]
```
6. 一般不应该, 在实地址模式下 BP 指向堆栈段。
7. 一般是可以的 (在平坦内存模式下堆栈和数据使用同一个段)。

9.5 整数数组的查找和排序

1. $n - 1$ 次。
2. $n - 1$ 次。
3. 否, 每次减 1。
4. $T(5000) - 0.5 \times 10^2$ 。
5. $(\log_2 128) + 1 = 8$ 。
6. $(\log_2 n) + 1$ 。
7. EDX 和 EDI 已经比较过了。
8. 这时可以把每条 JMP L4 指令改为 JMP L1。

第10章 结构和宏

10.1 结构

1. STRUCT 伪指令的作用是定义结构。在过程间传递大量数据时结构是必需的。

2. 结构定义:

```
MyStruct STRUCT
    field1 WORD ?
    field2 DWORD 20 DUP(?)
MyStruct ENDS
```

3. temp1 MyStruct <>

4. temp2 MyStruct <0>

5. temp3 MyStruct <, 20 DUP(0)>

6. array MyStruct 20 DUP(<>)

7. mov ax,array.field1

8. 代码示例:

```
mov esi,OFFSET array
add esi,3 * (TYPE myStruct)
mov (MyStruct PTR[esi]).field1,ax
```

9. 82。

10. 82。

11. TYPE MyStruct.field2 (或: SIZEOF Mystruct.field2)

12. 多个答案:

a. 是

b. 否

c. 是

d. 是

e. 否

13. 代码示例:

```
.data
time SYSTEMTIME <>
.code
mov ax,time.wHour
```

14. 代码示例:

```
myShape Triangle < <0,0>, <5,0>, <7,6> >
```

15. 代码示例 (初始化一个 triangle 结构数组):

```
.data
ARRAY_SIZE = 5
triangles Triangle ARRAY_SIZE DUP(<>)
.code
    mov     ecx,ARRAY_SIZE
    mov     esi,0
L1:  mov     eax,11
    call    RandomRange
    mov     triangles[esi].Vertex1.X, ax
    mov     eax,11
    call    RandomRange
```



```

mov     triangles[esi].Vertex1.Y, ax
add     esi,TYPE Triangle
loop    L1

```

10.2 宏

1. 错。
2. 对。
3. 带参数的宏复用起来更加容易一些。
4. 错。
5. 对。
6. 错。
7. 是为了允许在同一程序中多次使用的宏内定义标号。
8. ECHO 伪指令 (或者%OUT 操作符, 在本章后面讲述)。
9. 代码示例:

```

mPrintChar MACRO char,count
LOCAL temp
.data
temp BYTE count DUP(&char),0
.code
    push     edx
    mov     edx,OFFSET temp
    call    WriteString
    pop     edx
ENDM

```

10. 代码示例:

```

mGenRandom MACRO n
    mov     eax,n
    call    RandomRange
ENDM

```

11. mPromptInteger:

```

mPromptInteger MACRO prompt,returnVal
    mWrite prompt
    call    ReadInt
    mov     returnVal,eax
ENDM

```

12. 代码示例:

```

mWriteAt MACRO X,Y,literal
    mGotoxy X,Y
    mWrite literal
ENDM

```

13. 代码示例:

```

mWriteStr namePrompt
1    push     edx
1    mov     edx,OFFSET namePrompt
1    call    WriteString
1    pop     edx

```

14. 代码示例:

```

mReadStr customerName
1   push    ecx
1   push    edx
1   mov     edx,OFFSET customerName
1   mov     ecx,(SIZEOF customerName) - 1
1   call    ReadString
1   pop     edx
1   pop     ecx

```

15. 代码示例:

```

;-----
mDumpMemx MACRO varName
;
; Displays a variable in hexadecimal, using the
; variable's attributes to determine the number
; of units and unit size.
;-----
    push    ebx
    push    ecx
    push    esi
    mov     esi,OFFSET varName
    mov     ecx,LENGTHOF varName
    mov     ebx,TYPE varName
    call    DumpMem
    pop     esi
    pop     ecx
    pop     ebx
ENDM
; Sample calls:

.data
array1 BYTE 10h,20h,30h,40h,50h
array2 WORD 10h,20h,30h,40h,50h
array3 DWORD 10h,20h,30h,40h,50h
.code
mDumpMemx array1
mDumpMemx array2
mDumpMemx array3

```

10.3 条件汇编伪指令

1. IFB 伪指令用于检查宏参数是否为空。
2. IFIDN 伪指令比较两个文本值是否相同，如果相同则返回真，比较是大小写敏感的。
3. EXITM。
4. IFIDNI 是 IFIDN 的大小写敏感的版本。
5. 如果符号已经定义，IFDEF 返回真。
6. ENDIF。
7. 代码示例:

```

mWriteLn MACRO text:=<" ">
    mWrite text
    call CrLf
ENDM

```

8. 关系操作符列表:

```

LT   小于
GT   大于
EQ   等于

```

NE 不等于
LE 小于等于
GE 大于等于

9. 代码示例:

```
mCopyWord MACRO intVal
    IF (TYPE intVal) EQ 2
        mov ax,intVal
    ELSE
        ECHO Invalid operand size
    ENDM
```

10. 代码示例:

```
mCheck MACRO Z
    IF Z LT 0
        ECHO **** Operand Z is invalid ****
    ENDF
ENDM
```

11. 替换操作符 (&) 解析宏内对参数名的引用有歧义的情形。

12. 特殊文本字符操作符 (!) 强制预处理器把预定义的操作符作为普通字符对待。

13. 展开操作符 (%) 展开文本宏或把常量表达式转换为其文本表示形式。

14. 代码示例:

```
CreateString MACRO strVal
.data
temp BYTE "Var&strVal",0
.code
ENDM
```

15. 代码示例:

```
mLocate -2,20
;(no code generated because xval < 0)

mLocate 10,20
1 mov bx,0
1 mov ah,2
1 mov dh,20
1 mov dl,10
1 int 10h
mLocate col,row
1 mov bx,0
1 mov ah,2
1 mov dh,row
1 mov dl,col
1 int 10h
```

10.4 定义重复块

1. WHILE 伪指令基于一个布尔表达式重复一个语句块。
2. REPEAT 伪指令基于一个计数器的值重复一个语句块。
3. FOR 伪指令通过遍历一个符号列表中的每个符号来重复语句块。
4. FORC 伪指令通过遍历一个字符串中的每个字符来重复语句块。
5. FORC。
6. 代码示例:

```

BYTE 0,0,0,100
BYTE 0,0,0,20
BYTE 0,0,0,30

```

7. 代码示例:

```

mRepeat MACRO 'X',50
    mov cx,50
??0000: mov ah,2
    mov dl,'X'
    int 21h
    loop ??0000
mRepeat MACRO AL,20
    mov cx,20
??0001: mov ah,2
    mov dl,AL
    int 21h
    loop ??0001

mRepeat MACRO byteVal,countVal
    mov cx,countVal
??0002: mov ah,2
    mov dl,byteVal
    int 21h
    loop ??0002

```

8. 如果我们查看链表数据(在列表文件中),很明显,每个 ListNode 的 NextPtr 域(第二个节点的地址)总是等于 00000008:

Offset	ListNode
00000000	00000001 NodeData 00000008 NextPtr
00000008	00000002 NodeData 00000008 NextPtr
00000010	00000003 NodeData 00000008 NextPtr
00000018	00000004 NodeData 00000008 NextPtr
00000020	00000005 NodeData 00000008 NextPtr
00000028	00000006 NodeData 00000008 NextPtr

在教材中曾经给出过提示:地址计数器的值总是等于链表第一个节点的地址。

第11章 MS-Windows 程序设计

11.1 Win32 控制台编程

1. /SUBSYSTEM:CONSOLE。
2. 对。
3. 错。
4. 错。
5. 对。
6. BOOL = byte, COLORREF = DWORD, HANDLE = DWORD, LPSTR = PTR BYTE, WPARAM = DWORD。

7. GetStdHandle。
8. ReadConsole。
9. 来自 11.1.4 节 ReadConsole.asm 程序的例子:

```
INVOKE ReadConsole, stdInHandle, ADDR buffer,
      BufSize - 2, ADDR bytesRead, 0
```

10. COORD 结构包含了以字符为单位的 X, Y 的屏幕坐标。
11. 来自 11.1.5 节的 Console1.asm 程序的例子:

```
INVOKE WriteConsole,
      consoleHandle,           ; console output handle
      ADDR message,           ; string pointer
      messageSize,            ; string length
      ADDR bytesWritten,      ; returns num bytes written
      0,                      ; not used
```

12. 读取输入文件时调用 CreateFile 函数:

```
INVOKE CreateFile,
      ADDR filename,          ; ptr to filename
      GENERIC_READ,           ; access mode
      DO_NOT_SHARE,           ; share mode
      NULL,                   ; ptr to security attributes
      OPEN_EXISTING,          ; file creation options
      FILE_ATTRIBUTE_NORMAL,   ; file attributes
      0,                      ; handle to template file
```

13. 创建新文件时调用 CreateFile 函数:

```
INVOKE CreateFile,
      ADDR filename,
      GENERIC_WRITE,
      DO_NOT_SHARE,
      NULL,
      CREATE_ALWAYS,
      FILE_ATTRIBUTE_NORMAL,
      0
```

14. 调用 ReadFile:

```
INVOKE ReadFile,              ; read file into buffer
      fileHandle,
      ADDR buffer,
      bufSize,
      ADDR byteCount,
      0
```

15. 调用 WriteFile:

```
INVOKE WriteFile,            ; write text to file
      fileHandle,            ; file handle
      ADDR buffer,           ; buffer pointer
      bufSize,               ; number of bytes to write
      ADDR bytesWritten,     ; number of bytes written
      0,                     ; overlapped execution flag
```

16. SetFilePointer。
17. SetConsoleTitle。
18. SetConsoleScreenBufferSize。
19. SetConsoleCursorInfo。

20. SetConsoleTextAttribute。
21. WriteConsoleOutputAttribute。
22. Sleep。

11.2 编写 Windows 图形界面应用程序

注意：大部分问题都可以通过查看本书附带代码中的包含文件 GraphWin.inc 得到解答。

1. POINT 结构包含了两个域 ptX 和 ptY，描述了屏幕上点的 X 和 Y（像素）坐标。
2. WNDCLASS 结构定义了一个窗口类，程序中每个窗口都必须属于一个窗口类，每个程序都应该为其主窗口创建一个窗口类，窗口类是在主窗口显示之前由操作系统注册的。
3. lpfnWndProc 是一个指向应用程序函数的指针，该函数接收并处理用户引发的事件消息。
4. style 域是不同风格选项的组合，如 WS_CAPTION 和 WS_BORDER 等，这些选项控制着窗口风格和行为。
5. hInstance 存放着当前程序实例的句柄，每个 MS-Windows 下运行的程序在装入内存时操作系统都自动为其分配一个实例句柄。
6. （调用 CreateWindowEx 的例子程序见 11.2.6 节。）

CreateWindowEx 的函数原型在 GraphWin.inc 文件中：

```
CreateWindowEx Proto,
    classexWinStyle:DWORD,
    className:PTR BYTE,
    winName:PTR BYTE,
    winStyle:DWORD,
    X:DWORD,
    Y:DWORD,
    rWidth:DWORD,
    rHeight:DWORD,
    hWndParent:DWORD,
    hMenu:DWORD,
    hInstance:DWORD,
    lpParam:DWORD
```

其中的第 4 个参数 winStyle 确定了窗口的风格属性，在 11.2.6 节的 WinApp.asm 程序中，在调用 CreateWindowEx 时，传递了下面的预定义风格常量：

```
MAIN_WINDOW_STYLE = WS_VISIBLE + WS_DLGMFRAME + WS_CAPTION
                    + WS_BORDER + WS_SYSMENU + WS_MAXIMIZEBOX + WS_MINIMIZEBOX
                    + WS_THICKFRAME
```

这些常量确定的窗口是可见的，包含对话消息框架、一个标题栏、边、系统菜单、最小化图标以及一个粗边框。

7. 调用 MessageBox：

```
INVOKE MessageBox, hMainWnd, ADDR GreetText,
    ADDR GreetTitle, MB_OK
```

8. 从下面任选两个（来自 GraphWin.inc）：

```
MB_OK, MB_OKCANCEL, MB_ABORTRETRYIGNORE, MB_YESNOCANCEL, MB_YESNO,
MB_RETRYCANCEL, MB_CANCELTRYCONTINUE
```

9. 图标常量（从中任选两个）：

```
MB_ICONHAND, MB_ICONQUESTION, MB_ICONEXCLAMATION, MB_ICONASTERISK
```

10. WinMain 执行的任务如下（从中任选三个）：

获取当前程序的句柄。

加载程序的图标和鼠标光标。

注册程序的主窗口类并确定处理窗口事件消息的过程。

创建主窗口。

显示并更新主窗口。

开始一个循环接收并分发消息。

11. WinProc 过程接收并处理所有与窗口相关的事件消息, 它解码每条消息, 如果识别了该消息, 则执行与消息相关的面向应用的 (或应用特定的) 任务。
12. 处理了下面的消息:
 - WM_LBUTTONDOWN, 在用户按下鼠标左键时产生。
 - WM_CREATE, 表明主窗口刚刚创建。
 - WM_CLOSE, 表明程序的主窗口将要关闭。
13. ErrorHandler 过程是可选的, 在程序主窗口的创建和注册期间, 如果系统报告了错误就会被调用。
14. 消息对话框在程序主窗口出现之前显示。
15. 消息对话框在主窗口关闭之前出现。

11.3 动态内存分配

1. 动态内存分配。
2. 通过 EAX 返回程序现有的默认堆的 32 位整数句柄值。
3. 从堆中分配一块内存。
4. HeapCreate 的例子:

```
HEAP_START = 2000000          ; 2 MB
HEAP_MAX   = 400000000        ; 400 MB
.data
hHeap HANDLE ?                ; handle to heap
.code
INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX
```

5. 传递一个指向内存块的指针 (以及堆的句柄)。

11.4 IA-32 内存管理

1. (a) 多任务允许多个程序 (或任务) 同时运行, 处理器把时间分给在运行的所有程序。
(b) 分段提供了一种方法, 把内存段相互隔离开来, 这使得多个程序能够同时运行但不会相互影响。
2. (a) 段选择子是存储在段寄存器 (CS, DS, ES, SS, FS, GS) 中的一个 16 位的值。
(b) 逻辑地址是段选择子和一个 32 位偏移的组合。
3. 对。
4. 对。
5. 错。
6. 错。
7. 线性地址指向内存中的一个位置, 是 0~FFFFFFFFh 之间的一个 32 位整数。如果分页特性被禁用, 线性地址就是物理地址。
8. 在允许分页时, 处理器把 32 位的线性地址翻译成 32 位的物理地址。线性地址被分成三

个域：指向页目录的指针、指向页表的指针和在页中的偏移。

9. 线性地址被自动翻译成 32 位的物理地址。
10. 分页使得计算机同时运行物理内存无法同时容纳的多个程序成为可能，处理器初始时只把程序的一部分载入内存，其他部分保留在磁盘上，在需要时再载入（换入）内存。
11. LDTR 寄存器。
12. GDTR 寄存器。
13. 1 个。
14. 多个（每个任务或程序都有自己的局部描述符表）。
15. 从下面任选 4 个：基地址、特权级、段类型、段存在标志、粒度标志、段界限。
16. 页目录、页表和页。
17. 线性地址的页表域（参见图 11.4）。
18. 线性地址的偏移地址域（参见图 11.4）。

第 12 章 高级语言接口

12.1 简介

1. 一种语言使用的命名约定是指语言中过程及变量命名的规则或特性。
2. 微型、小型、压缩、中型、大型、巨型。
3. 不能。因为过程名不能被链接器找到。
4. 是的，这是由于调用约定确定了调用是近程的还是远程的。近程调用只在堆栈上压入 16 位的返回偏移地址，远程调用在堆栈上压入 32 位的段/偏移返回地址。
5. 由于 C/C++ 是大小写敏感的，因此它们只执行具有相同大小写形式的过程名的调用。
6. 是的，许多语言都特别规定在过程调用之间必须保护 EBP（BP）、ESI（SI）以及 EDI（DI）寄存器。

12.2 内联汇编代码

1. 内联汇编代码是直接插入到高级语言程序中的汇编语言源代码，C++ 的 inline 关键字则是要求 C++ 编译器直接在编译后的程序代码中在调用该函数的地方直接插入函数体，目的是为了减少函数调用和从函数返回的时间开销（回答本问题需要了解一些本书未包含的 C++ 语言的知识）。
2. 编写内联汇编代码的主要优点是其简单性，不需要担心额外的链接、命名以及参数传递协议等问题；其次，内联汇编代码避免了通常调用外部过程时过程调用和从过程返回的开销，因此执行得更快些。
3. 注释的例子（任选两种）：

```
mov esi,buf           ; initialize index register
mov esi,buf           // initialize index register
mov esi,buf           /* initialize index register */
```
4. 是。
5. 是。
6. 否。
7. 否。
8. 由于 __fastcall 允许编译器使用通用寄存器保存临时变量的值，因此在内联汇编代码中修

改通用寄存器有可能会导致 bug。

9. 使用 LEA 指令。

10. LENGTH 操作符返回 DUP 操作符定义的数组中元素的数目。例如下面的代码中放入 EAX 中的值是 20:

```
myArray DWORD 20 DUP(?), 10, 20, 30
.code
mov eax, LENGTH myArray          ; 20
```

(注意: 第 4 章中介绍的 LENGTHOF 操作符对于 myArray 将返回 23。)

11. SIZE 操作符返回 TYPE(4) * LENGTH 的结果。

12.3 在保护模式下与 C++ 程序链接

1. 必须使用 extern “C” 关键字。
2. Irvine32 库使用的是 STDCALL 调用约定, 这与 C/C++ 使用的 C 调用约定是不同的, 其最大区别在于函数调用之后如何清理堆栈。
3. 浮点值通常在函数返回之前压入处理器的浮点堆栈。
4. 通过 AX 返回 short int。
5. printf PROTO C, pString:PTR BYTE, args:VARARG。
6. x 最后压栈。
7. 阻止 C++ 编译器修饰 (改变) 外部过程的名字, 名字修饰通常是那些允许函数重载 (也就是允许多个函数有同样的名字) 的程序设计语言提供的一种命名机制。
8. 如果发生了名字修饰, 汇编语言中调用的过程名和 C++ 编译器生成的外部函数名就会不同。汇编器没有关于 C++ 编译器名字修饰规则的任何知识, 这是可以理解的。
9. 几乎没有什么不同, 在操纵数组时使用下标和使用指针一样高效。

12.4 在实地址模式下与 C/C++ 程序链接

1. Borland C++ 调用的汇编语言过程必须保护 BP, DS, SS, SI, DI 和方向标志的值。
2. INT = 2, enum = 1, float = 4, double = 8。
3. Mov eax, [bp + 6]。
4. rot eax, 8 语句循环移出 EAX 的最低位, 当产生小的随机数时可防止循环模式。

第 13 章 16 位 MS-DOS 程序设计

13.1 MS-DOS 和 IBM-PC

1. 9FFFh。
2. 中断向量表。
3. 00400h。
4. BIOS。
5. 假设程序名为 myProg.exe, 下面的命令行把输出重定向到默认打印机:
myProg > prn
6. LPT1。
7. 中断服务例程是能够完成下列任务的操作系统过程: (1) 为应用程序提供基本的服务; (2) 处理硬件事件。更多细节请参见 16.4 节。
8. 标志压栈。

9. 参见 13.1.4 节的 4 个步骤。
10. 中断处理过程执行 IRET 指令, 返回到应用程序继续执行。
11. 10h。
12. 1Ah。
13. $21h * 4 = 0084h$ 。

13.2 MS-DOS 功能调用 (INT 21h)

1. AH。
2. 功能 4Ch。
3. 功能 2 和 6 都能显示一个字符。
4. 功能 9。
5. 功能 40h。
6. 功能 1 和 6。
7. 功能 3Fh。
8. 功能 2Ah 和 2Bh。要显示时间, 可以调用本书链接库附带的 WriteDec 过程, 该过程调用功能 2 在控制台上显示数字。(细节参见 Irvine16.asm, 在 \Examples\Lib16 目录下)。
9. 功能 2Bh (设置系统日期) 和 2Dh (设置系统时间)。
10. 功能 6。

13.3 标准 MS-DOS 文件 I/O 服务

1. 设备句柄: 0 = 键盘 (标准输入), 1 = 控制台 (标准输出), 2 = 错误输出, 3 = 辅助设备 (异步), 4 = 打印机。
2. 进位标志。
3. 功能 716Ch 的参数:

```
AX = 716Ch
BX = access mode (0 = read, 1 = write, 2 = read/write)
CX = attributes (0 = normal, 1 = read only, 2 = hidden,
3 = system, 8 = volume ID, 20h = archive)
DX = action (1 = open, 2 = truncate, 10h = create)
DS:SI = segment/offset of filename
DI = alias hint (optional)
```

4. 打开一个已存在的文件进行输入:

```
.data
infile BYTE "myfile.txt",0
inHandle WORD ?
.code
    mov     ax,716Ch                ; extended create or open
    mov     bx,0                    ; mode = read-only
    mov     cx,0                    ; normal attribute
    mov     dx,1                    ; action: open
    mov     si,OFFSET infile
    int     21h                     ; call MS-DOS
    jc      quit                    ; quit if error
    mov     inHandle,ax
```

5. 从文件中读取一个二进制数组最好使用 INT 21h 的功能 3Fh, 需要下面的参数:

```
AH = 3Fh
BX = open file handle
CX = maximum bytes to read
DS:DX = address of input buffer
```

6. 在调用 INT 21h 之后, 比较 AX 中的返回值和在调用该功能之前放入 CX 寄存器中的值是否相同, 如果 AX 中的值较小, 则到达了文件末尾。
7. 惟一的区别是 BX 中的值。在从键盘读的时候, BX 被设为键盘句柄 (0), 在从文件中读的时候, BX 被设为文件句柄。
8. 功能 42h。
9. 代码示例 (BX 中已经包含了文件句柄):

```
mov ah,42h                ; move file pointer
mov al,0                  ; method: offset from beginning
mov cx,0                  ; offsetHi
mov dx,50                 ; offsetLo
int 21h
```

第 14 章 磁盘基础知识

14.1 磁盘存储系统

1. 对。
2. 错。
3. 柱面。
4. 对。
5. 512。
6. 是为了进行快速访问, 因为柱面靠得越近, 读取时读写磁头要移动的距离越短。
7. 读写磁头必须跳过其他的柱面, 这会浪费时间并且增加出错的概率。
8. 卷。
9. 读写磁头在磁道之间移动所需的平均时间。
10. 在磁盘的表面标记物理扇区。
11. 磁盘分区表和一段定位分区引导扇区并且运行另外一个加载操作系统的程序的程序。
12. 1 个。
13. 系统。

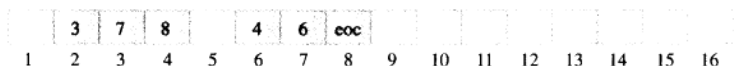
14.2 文件系统

1. 对。
2. 错, 在磁盘目录中。
3. 错 (包括 NTFS 在内的所有的系统存储文件时都要求使用至少一个簇)。
4. 错。
5. 错。
6. 4 GB (如表 14.1 所示)。
7. FAT32 和 NTFS。
8. NTFS。
9. NTFS。
10. NTFS。
11. NTFS。
12. 8 GB。
13. 引导记录、文件分配表、根目录、数据区。

14. 在引导记录的偏移 0Dh 处存储着该信息。
15. 两个 8 KB 的簇总共需要 16 384 个字节，浪费的字节数为 16 384 - 8200，也就是 8184 个字节。
16. 无答案。

14.3 磁盘目录

1. 对。
2. 错（称为根目录）。
3. 错（包含起始簇号）。
4. 对。
5. 32。
6. 文件名、扩展名、属性、时间戳、日期戳、起始簇号、文件大小。
7. 状态字节及描述参见表 14.5。
8. 位 0~4 = 秒，位 5~10 = 分钟，位 11~15 = 小时。
9. 目录项的第一个字节是 4xh，其中的 x 表示文件名使用的长文件名目录项的数目。
10. 两个。
11. 实际有三个新的域：最后访问时间、创建日期、创建时间。
12. 文件分配表中的链接：



14.4 读写磁盘扇区（7305h）

1. 对。
2. 错（该功能只能在实地址模式下运行）。
3. 参数：
 - AX: 7305h
 - DS:BX: Segment/offset of a DISKIO structure variable
 - CX: 0FFFFh
 - DL: Drive number (0 = default, 1 = A, 2 = B, 3 = C, etc.)
 - SI: Read/write flag
4. INT 10h 能够显示特殊的 ASCII 图形字符而不会把它们解释成控制代码（如制表和回车）。
5. 如果功能 7305h 不能读取指定的扇区，则进位标志置位，程序显示一条错误消息（一定要注意不能在 Windows NT/2000/XP 下测试这个程序）。

14.5 系统级文件功能调用

1. 功能 7303h。
2. 功能 7303h。
3. 功能 39h（创建子目录）和功能 3Bh（设置当前目录）。
4. 功能 7143h（获取和设置文件属性）。

第 15 章 BIOS 程序设计

15.1 简介

无习题。

15.2 INT 16h 键盘中断

1. INT 16h 最适合。
2. 在位置 0040:001E 处的键盘缓冲区中。
3. INT 9h 读取键盘输入, 获取键盘扫描码并产生对应的 ASCII 码, 然后把扫描码和 ASCII 码都插入键盘缓冲区中。
4. 功能 05h。
5. 功能 10h。
6. 功能 11h 检查缓冲区并返回等待的按键 (如果有的话)。
7. 否。
8. 功能 12h。
9. 位 4 (参见表 15.2)。
10. 代码示例:

```
L1:  mov ah,12h                ; get keyboard flags
      int 16h
      test al,100h            ; Ctrl key down?
      jz L1                   ; no: repeat the loop
      ; At this point, the Ctrl key has been pressed
```

11. 要检查其他按键, 在循环中已有的 CMP 和 JE 指令后添加更多的 CMP 和 JE 指令。假设想要检查 ESC、F1 和 Home 按键, 代码如下:

```
L1:  .
      .
      cmp ah,1                ; ESC key's scan code?
      je quit                 ; yes: quit
      cmp ah,3Bh              ; F1 function key?
      je quit                 ; yes: quit
      cmp ah,47h              ; Home key?
      je quit                 ; yes: quit
      jmp L1                  ; no: check buffer again
```

15.3 INT 10h 视频程序设计

1. MS-DOS 层、BIOS 层和直接视频显示。
2. 直接视频显示。
3. 在 MS-Windows 下有两种方法切换到全屏模式: (1) 为程序的可执行文件创建一个快捷方式, 打开快捷方式的属性对话框, 选择屏幕标签, 然后选择全屏幕模式选项; (2) 从开始菜单中打开一个命令行窗口, 然后按下 Alt-Enter 键切换到全屏模式。使用 CD (改变目录) 命令进入程序可执行文件所在的目录, 输入程序的名字运行程序。Alt-Enter 是一个开关, 再次按下它就可以返回到窗口模式。
4. 模式 3 (彩色, 80×25)。
5. 字符的 ASCII 码和属性字节 (总共两个字节)。
6. 红、绿、蓝以及亮度。
7. 背景: 位 4~7, 前景: 位 0~3。
8. 功能 02h。
9. 功能 06h。
10. 功能 09h。
11. 功能 01h。

12. 功能 00h。
13. AH = 2, DH = 行, DL = 列, BH = 视频页。
14. 有两种方法: (1) 使用 INT 10h 的功能 01h 把光标顶部的扫描线设为非法值; (2) 使用 INT 10h 的功能 02h 把光标定位在显示的行列范围之外。
15. AH = 6, AL = 要卷动的行数, BH = 卷动空出行的属性, CH & CL = 窗口左上角的坐标, DH & DL = 窗口右下角的坐标。
16. AH = 9, AL = 字符的 ASCII 码, BH = 视频页, BL = 属性, CX = 重复次数。
17. 功能 10h 的子功能 03h (AH 设为 10h, AL 设为 03h)。
18. AH = 06h, AL = 0。
19. 屏幕上的每个像素点都是由三种颜色构成的: 红、绿、蓝。狗是色盲, 因此它们看不到彩色组成的像素, 我曾经试过在屏幕上显示一只猫的图片, 但我的狗看起来似乎没有注意到这幅图片。

15.4 使用 INT 10h 绘图

1. 功能 0Ch。
2. AH = 0Ch, AL = 像素值, BH = 视频页, CX = X 坐标, DX = Y 坐标。
3. 非常慢。
4. 代码示例:


```
mov ah,0                ; set video mode
mov al,11h              ; to mode 11h
int 10h                 ; call the BIOS
```
5. 模式 6AH。
6. 公式: $sx = (sOrigX + X)$ 。
7. a. (350,150) b. (375,225) c. (150,400)

15.5 内存映射图形

1. 错 (每个字节对应于 1 个像素)。
2. 对。
3. 模式 13h 把每个像素的整数值映射到一张称为调色板的表中。
4. 像素的彩色索引指明了在屏幕上绘制像素的时候要使用调色板中的哪种颜色。
5. 调色板中的每个表项都由三个 0~63 之间的独立整数值构成, 这称为 RGB (红、绿、蓝)。调色板的表项 0 控制着屏幕的背景色。
6. (20,20,20)。
7. (63,63,63)。
8. (63,0,0)。
9. 代码示例:

```
; Set screen background color to bright green.
mov dx,3c8h                ; video paletter port
mov al,0                   ; index 0 (background color)
out dx,al
mov dx,3c9h                ; colors go to port 3C9h
mov al,0                   ; red
out dx,al
mov al,63                  ; green (intensity = 63)
out dx,al
mov al,0                   ; blue
out dx,al
```

10. 代码示例:

```

; Set screen background color to white
mov dx,3c8h                ; video paletter port
mov al,0                   ; index 0 (background color)
out dx,al
mov dx,3c9h                ; colors go to port 3C9h
mov al,63                  ; red = 63
out dx,al
mov al,63                  ; green = 63
out dx,al
mov al,63                  ; blue = 63
out dx,al

```

(如果想要减少代码数量,上面例子中最后两条 MOV 指令是可以省略的。)

15.6 鼠标程序设计

1. 功能 0。

2. 代码示例:

```

mov ax,0                   ; reset mouse
int 33h                   ; call the BIOS
cmp ax,0                   ; mouse not available?
je MouseNotAvailable      ; yes: show error message

```

3. 功能 1 和功能 2。

4. 代码示例:

```

mov ax,2                   ; hide mouse pointer
int 33h

```

5. 功能 3

6. 代码示例:

```

mov ax,3                   ; get mouse position and status
int 33h
mov mouseX,cx
mov mouseY,dx

```

7. 功能 4。

8. 代码示例:

```

mov ax,4                   ; set mouse position
mov cx,100                 ; X-value
mov dx,400                 ; Y-value
int 33h

```

9. 功能 5。

10. 示例代码:

```

mov ax,5                   ; get button press information
mov bx,0                   ; button ID for left button
int 33h
test ax,1                  ; left button currently down?
jne Button1                ; yes: jump to label

```

实现笔记: 这个函数能够告诉你当前是否有按钮按下,但如果只想获取最后按下按钮的坐标,在上面的代码中就没有必要使用 TEST 指令了。

11. 功能 6。

12. 代码示例:

```

mov     ax,6                ; get button release information
mov     bx,1                ; button ID
int     33h
test    ax,2                ; right button released?
jz      skip                ; no - skip
mov     mouseX,cx           ; yes: save coordinates
mov     mouseY,dx
skip:

```

13. 代码示例:

```

mov     ax,8                ; set vertical limits
mov     cx,200              ; lower limit
mov     dx,400              ; upper limit
int     33h

```

14. 代码示例:

```

mov     ax,7                ; set horizontal limits
mov     cx,300              ; lower limit
mov     dx,600              ; upper limit
int     33h

```

15. 假设字符单元格是 8×8 像素的, X 和 Y 的坐标值将是 $(8*20)$, $(8*10)$, 因此单元格在位置 $(160, 80)$ 处。
16. 单元格的左上角坐标是 $(8*22)$, $(8*15)$, 因此要把鼠标定位在单元格的中央, 这两个值都要加 4, 因此最后的答案是 $(180, 124)$ 。
17. 鼠标是由斯坦福研究院的 Douglas Engelbat 于 1963 年发明的 (来源: http://en.wikipedia.org/wiki/Computer_mouse)。

第 16 章 高级 MS-DOS 程序设计

16.1 简介

无习题。

16.2 定义段

1. 声明段的开始和结束。
2. 返回代码标号或数据标号的段地址。
3. ASSUME 伪指令使得汇编器能够生成在运行时计算编译期标号或变量地址的代码, 伪指令如:

```
assume DS:myData
```

告诉汇编器 “从现在开始, 引用的所有对数据标号 (通过 DS) 都位于 myData 段内”。

4. BYTE, WORD, DWORD, PARA 和 PAGE。
5. PRIVATE, PUBLIC, MEMORY, STACK, COMMON 和 AT。
6. DWORD。
7. 组合类型通知链接器如何组合同名的段。
8. 使用 AT 组合类型, 下面的伪指令定义了段值为 0040h 的段:

```
bios SEGMENT AT 40h
```

9. 段的类型为不同名字的段提供了另外一种组合方式, 同样类型的段尽管在源码中的位置

可能是随意的,但链接器会把同样类型的段放在相邻的位置。

10. 代码示例:

```
mov al,es:[di]
```

11. 第三个段也从地址 1A060h 处开始。

16.3 程序的运行时结构

1. 命令行处理器检查在当前目录中是否有 COM 后缀的文件,如果有,就执行它,如果未找到匹配文件,后续的详细步骤请参见 16.3 节中的描述。
2. 否。
3. 应用程序加载至最低的 640 KB 的内存,它们是暂留程序,在执行完成之后,将被卸载出内存。
4. 程序段前缀。
5. 在程序段前缀的偏移 2Ch 处。
6. COM 程序是只有一个段的 MS-DOS 程序,它只是加载到内存的代码的二进制映像。
7. 小型内存模式。
8. *IT*。
9. 64 KB。
10. 效率不高,这是由于即使是最小的 COM 程序也要使用整个 64 KB 的内存段。
11. 1 个。
12. 所有的段寄存器都初始化为指向程序 PSP 的基地址,COM 程序加载在内存中已有程序的后面的第一个可用段地址处。
13. ORG 伪指令为紧跟其后的标号或代码赋予一个特殊的偏移地址,所有后面的标号的地址都在该偏移地址的基础上进行计算,例如 COM 程序,在程序的开始总会有一条 ORG 100h 伪指令,因此 COM 程序的第一条可执行指令在段内偏移地址 100h 处。
14. 可装载程序模块。
15. DS 和 ES 指向程序的程序段前缀区域。
16. 除非程序的 EXE 头限制了最大可分配内存的数量,MS-DOS 在程序首次加载的时候自动给程序分配所有可用的内存。
17. exehdr 程序显示程序内存使用的统计信息,同时允许修改 EXE 头的许多参数设置。
18. 运行 exehdr 程序,向它传递 EXE 文件的名字,程序运行后显示的最后一行中包含了重定位项的数目。

16.4 中断处理

1. 在屏幕上显示一条消息 “Abort, retry, or ignore?” 并结束当前程序。
2. 指向中断处理程序的 32 位的段/偏移地址。
3. 在地址 0000:0040h 处,这是由于 0040h 等于 $10h * 4$ 。
4. 8259 可编程中断控制芯片。
5. CLI (清除中断标志) 指令。
6. STI (设置中断标志) 指令。
7. IRQ 0 的优先级最高。
8. 在文件被创建之前,这是由于键盘中断 (IRQ 1) 比磁盘中断 (IRQ 14) 的优先级更高的缘故。

9. INT 9h。
10. 中断处理过程结束时的 IRET 指令把控制权返回给中断发生时正在执行的代码。
11. 功能 25h 和 35h。
12. 中断处理过程可以是任何处理中断的过程。它可以在应用程序启动时加载,然后在应用程序退出时卸载。相反,内存驻留中断处理过程即使在安装它的程序结束后仍然保留在内存中。内存驻留程序不一定就是中断处理程序。
13. 终止驻留程序(TSR)在退出时把自己的部分留在内存中,这是通过调用 INT 21h 的功能 31h 实现的。
14. 可以重启计算机或使用特殊的工具程序删除 TSR。
15. 可以在结束时不执行 IRET 指令,而是执行一条 JMP 指令跳转到前面已保存的中断向量表中的原来的处理程序的地址处。
16. 终止驻留程序(TSR)。
17. Ctrl + Alt + 右 Shift + Del。

第 17 章 浮点处理和指令编码

17.1 浮点二进制表示

1. 这是由于 $-127 + 127 = 0$, 如果以 -127 作为最小的指数,那么最小指数的倒数会发生溢出。
2. 这是由于 $+128$ 和指数的基数相加得到的值是负数。
3. 52 位。
4. 8 位。
5. $1101.01101 = 13/1 + 1/4 + 18 + 1/32$ 。
6. 这是由于十进制数 0.2 的二进制表示是无限循环的位模式。
7. $11011.0101 = 1.101101011 \times 2^4$ 。
8. $0000100111101.1 = 1.001111011 \times 2^{-8}$ 。
9. $+1110.011 = 1.110011 \times 2^{-3}$, 因此编码是 0 01111100 1100110000000000000000。
10. Quiet NaN 和 Signaling NaN。
11. $5/8 = 0.101$, 二进制。
12. $17/32 = 0.10001$, 二进制。
13. $+10.75 = +1010.11 = +1.01011 \times 2^3$, 编码为 0 10000010 0101100000000000000000。
14. $-76.0625 = -01001100.0001 = -1.0011000001 \times 2^{-6}$, 编码为:
1 10000101 001100000100000000000000
15. 根据数的符号,为正无穷或负无穷。

17.2 浮点单元

1. fld st(0)。
2. R0。
3. 可以从操作码寄存器、控制寄存器、状态寄存器、标记字寄存器、最后指令指针寄存器和最后数据指针寄存器中选出三个。
4. 二进制编码的实数。
5. 没有。
6. REAL10, 包含 80 个数据位。

7. 从堆栈中弹出 ST(0)。
8. FCHS。
9. 无操作数、m32fp、m64fp、堆栈寄存器。
10. FISUB 首先把整数源操作数转换成浮点数。
11. FCOM 或 FCOMP。
12. 代码示例:

```
fnstsw ax
lahf
```
13. FILD。
14. RC 域。
15. 1.010101101 近似到最近的偶数是 1.010101110。
16. -1.010101101 近似到最近的偶数是 -1.010101110。
17. 汇编指令:

```
.data
B REAL8 7.8
M REAL8 3.6
N REAL8 7.1
P REAL8 ?
.code
fld M
fchs
fld N
fadd B
fmul
fst P
```

18. 汇编语言代码:

```
.data
B DWORD 7
N REAL8 7.1
P REAL8 ?
.code
fld N
fsqrt
fiadd B
fst P
```

17.3 Intel 指令编码

1. (a) 8E (b) 8B (c) 8A (d) 8A (e) A2 (f) A3
2. (a) 8E (b) 8A (c) 8A (d) 8B (e) A0 (f) 8B
3. (a) D8 (b) D3 (c) 1D (d) 44 (e) 84 (f) 85
4. (a) 06 (b) 56 (c) 1D (d) 55 (e) 84 (f) 81
5. 机器语言字节:
 - a. 8E D8
 - b. A0 00 00
 - c. 8B 0E 01 00
 - d. BA 00 00
 - e. B2 02
 - f. BB 00 10